NAVAL POSTGRADUATE SCHOOL
Monterey, California

Engineering Automation for Reliable Software

Interim Progress Report (10/01/2000 – 09/30/2001)

by

Luqi

September 2001

NAVAL POSTGRADUATE SCHOOL
Monterey, California 93943-5000

RADM David R. Ellison
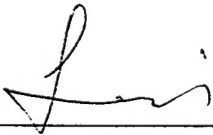Superintendent

Richard S. Elster
Provost

This report was prepared for U.S. Army Research Office and funded in part by the U.S. Army Research Office.
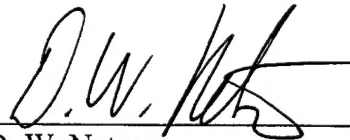
This report was prepared by:

Luqi
Professor, Computer Science

Reviewed by:

Luqi
Director, Software Engineering
Automation Center

Released by:

D. W. Netzer
Associate Provost and
Dean of Research

REPORT DOCUMENTATION PAGE

Form Approved
OMB NO. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE 9/30/2001	3. REPORT TYPE AND DATES COVERED Interim Progress Report 10/01/2000 – 09/30/2001

4. TITLE AND SUBTITLE Engineering Automation for Reliable Software – Interim Progress Report (10/01/1999 – 09/30/2000)	5. FUNDING NUMBERS 40473-MA-SP

6. AUTHOR(S)

Professor Luqi

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Software Engineering Automation Center, Naval Postgraduate School, Monterey, CA 93943	8. PERFORMING ORGANIZATION REPORT NUMBER NPS-SW-01-004

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) U. S. Army Research Office P.O. Box 12211 Research Triangle Park, NC 27709-2211	10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12 a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution unlimited.	12 b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

The objective of our effort is to develop a scientific basis for producing reliable software that is also flexible and cost effective for the DoD distributed software domain. This objective addresses the long term goals of increasing the quality of service provided by complex systems while reducing development risks, costs, and time. Our work focuses on "wrap and glue" technology based on a domain specific distributed prototype model. The key to making the proposed approach reliable, flexible, and cost-effective is the automatic generation of glue and wrappers based on a designer's specification. The "wrap and glue" approach allows system designers to concentrate on the difficult interoperability problems and defines solutions in terms of deeper and more difficult interoperability issues, while freeing designers from implementation details. Specific research areas for the proposed effort include technology enabling rapid prototyping, inference for design checking, automatic program generation, distributed real-time scheduling, wrapper and glue technology, and reliability assessment and improvement. The proposed technology will be integrated with past research results to enable a quantum leap forward in the state of the art for rapid prototyping.

14. SUBJECT TERMS Rapid Prototyping, Lightweight Inference, Automatic Program Generation, Distributed Real-time Scheduling, Wrapper and Glue, Reliability Assessment, Interoperability, Heterogeneous System Integration	15. NUMBER OF PAGES
	16. PRICE CODE

17. SECURITY CLASSIFICATION OR REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION ON THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL

Table of Contents

Interim Progress Report

Engineering Automation for Reliable Software

10/1/2000 – 9/30/2001

Luqi

Statement of the Problem Studied:

This project addresses the problem of how to produce reliable software that is also flexible and cost effective for the DoD distributed software domain. Current and future DoD software systems fall into two categories: information systems and warfighter systems. Both kinds of systems can be distributed, heterogeneous and network-based, consisting of a set of components running on different platforms and working together via multiple communication links and protocols.

Summary of Important Results:

We focused on "wrap and glue" technology based on a domain specific distributed prototype model. Glue and wrappers consists of software that bridges the interoperability gap between individual COTS/GOTS components. The key to making the proposed approach reliable, flexible, and cost-effective is the automatic generation of glue and wrappers based on a designer's specification. The proposed "wrap and glue" approach allows system designers to concentrate on the difficult interoperability problems and defines solutions in terms of deeper and more difficult interoperability issues, while freeing designers from implementation details. The objective of our research is to develop an integrated set of formal models and methods for system engineering automation. These results will enable building decision support tools for concurrent engineering. Our research addresses complex modular systems with embedded control software and real-time requirements.

Our longer-term goals are to construct an integrated set of software tools that can improve software quality and flexibility by automating a significant part of the process and providing substantial decision support for the aspects that cannot be automated. The resulting development environment should be adaptable to enable (1) maintaining integrated support in the presence of business process improvement, (2) incorporation of future improvements in engineering automation methods, and (3) specialization to particular problem domains.

In FY01, we investigated models and methods for solving the integration and interoperability problems in component-based distributed heterogeneous systems development.

Our work resulted in models and languages for specifying the architecture of distributed heterogeneous systems and components, as

1

well as technologies and tools to automate the integration of distributed heterogeneous software component via the automatic generation of glue and wrappers from specifications.

We developed an object-oriented model for an wrapper-based translator to resolve the representational differences between heterogeneous systems; an integrated development environment for users to create such models; methods for determining object correspondence during system integration; and the use of the Extensive Markup Language (XML) as a means for establishing interoperability between multiple DoD databases.

We also developed techniques for decision support for optimizing distributed object servers utilization, as well as the use software decoys to improve the security of distributed heterogeneous systems.

In addition, we investigated formal risk assessment models for the evolutionary software process. We formulated methods and tools to assess the risk and the duration of software projects automatically, based on measurements (requirements volatility, production team efficiency, and product complexity) that can be obtained early in the development process. The effectiveness of the models was validated by comparing the results of the models against data collected from 3 large real projects and 16 simulated projects.

We also worked with the US Army TACOM to develop formal models and methods to assess the maturity/risk of emerging software technologies and to assist managers to size the software technology infrastructure.

List of Publications:

M. Auguston, "Visual Meta-Programming Notation", *Proceedings of the 8th Monterey Workshop "Engineering Automation for Software Intensive System Integration" (Monterey Workshop 2001), Monterey, California, June 19-21, 2001.*

N. Cheng, V. Berzins, Luqi and S. Bhattacharya, "JAVA Wrappers for Automated Interoperability", *Lecture Notes in Computer Science*, Vol. 1966, Springer-Verlag, 2000, pp 45-64.

J. Ge, V. Berzins and M. Shing, "An agent-based, distributed prototyping system for software interoperability study", *Proceedings of the 13th International Conference on Computer Applications in Industry and Engineering of the International Society for Computers and Their Applications*, Honolulu, HI, USA, November 1-3, 2000, pp. 224-227.

J. Ge, V. Berzins and Luqi, "Computer Aided Prototyping in a Distributed Environment", *Proceedings of the International ICSC Congress on Intelligent Systems & Applications (ISA'2000)*, University of Wollongong, Australia, December 11-15, 2000.

J. Ge, B. Kin and V. Berzins, "A Software Agent Framework for Distributed Applications", *Proceedings of the 14th International Conference on Parallel and Distributed Computing Systems*, Dallas, TX, August 8-10, 2001.

2

O. Kiselyov, "Subclassing errors, OOP & Practically Checkable Rules to Prevent Them", *Proceedings of the 8th Monterey Workshop "Engineering Automation for Software Intensive System Integration" (Monterey Workshop 2001), Monterey, California, June 19-21, 2001.*

Luqi, V. Berzins and M. Shing, "The Use of Computer-Aided Prototyping for Reengineering Legacy Software", *Proceedings of the 8th Monterey Workshop "Engineering Automation for Software Intensive System Integration" (Monterey Workshop 2001), Monterey, California, June 19-21, 2001.*

Luqi, V. Berzins, J. Ge, M. Shing, M. Auguston, B. Bryant and B. Kin, "DCAPS - Architecture for Distributed Computer Aided Prototyping System", *Proceedings of the 12th IEEE International Workshop on Rapid System Prototyping (RSP2001),* Monterey, California, June 25-27, 2001, pp. 103-108.

J. Michael and R. Riehle, "Intelligent Software Decoys", *Proceedings of the 8th Monterey Workshop "Engineering Automation for Software Intensive System Integration" (Monterey Workshop 2001), Monterey, California, June 19-21, 2001.*

M. Murrah, C. Johnson and Luqi, "Enhancements and Extensions of Formal Models for Risk Assessment in Software Projects", *Proceedings of the 8th Monterey Workshop "Engineering Automation for Software Intensive System Integration" (Monterey Workshop 2001), Monterey, California, June 19-21, 2001.*

R. Raje, M. Auguston, B. Bryant, A. Olson and C. Burt, "A Unified Approach for the Integration of Distributed Heterogeneous Software Components", *Proceedings of the 8th Monterey Workshop "Engineering Automation for Software Intensive System Integration" (Monterey Workshop 2001), Monterey, California, June 19-21, 2001.*

W. Ray and V. Berzins, "Optimization of Distributed Object-Oriented Servers", *Proceedings of the 8th Monterey Workshop "Engineering Automation for Software Intensive System Integration" (Monterey Workshop 2001), Monterey, California, June 19-21, 2001.*

M. Saboe and Luqi, "A Software Technology Transition Engine", *Proceedings of the 8th Monterey Workshop "Engineering Automation for Software Intensive System Integration" (Monterey Workshop 2001), Monterey, California, June 19-21, 2001.*

P. Young, V. Berzins, J. Ge and Luqi, "Use of Object Oriented Model for Interoperability Wrapper-Based Translator for Resolving Representational Differences between Heterogeneous Systems", *Proceedings of the 8th Monterey Workshop "Engineering Automation for Software Intensive System Integration" (Monterey Workshop 2001), Monterey, California, June 19-21, 2001.*

Scientific Personnel:

Dr. Bret Michael, Associate Professor, NPS

Dr. Du Zhang, Visiting Professor, NPS.

Dr. Swapan Bhattacharya, National Research Council Research Associate

Dr. Jun Ge, National Research Council Research Associate

Dr. Mikhail Auguston, National Research Council Research Associate

Dr. Oleg Kiselyov, National Research Council Research Associate

Dr. Barrett Bryant, National Research Council Research Associate

Dr. Nabendu Chaki, Visiting Professor, NPS

Boon Kwang Kin, "A Simple Software Agents Framework for Building Distributed Applications", Master thesis, NPS, March 2001.

Eddie Davis, "Evaluation of the Extensive Markup Language (XML) as a Means for Establishing Interoperability Between Multiple DoD Databases", Master thesis, NPS, June 2001.

Robert Halle, "Extensible Markup Language (XML) Based Analysis and Comparison of Heterogeneous Databases", Master thesis, NPS, June 2001.

Craig Johnson and Robert Piirainen, "Application of the Nogeuria Risk Assessment Model to Real-Time Embedded Software Projects", Master thesis, NPS, June 2001.

Wayne Mandak and Charles Stowell, "Dynamic Assembly for System Adaptability Dependability and Assurance (DASADA) Project Analysis", Master thesis, NPS, June 2001.

Paul Nelson, "A Requirements Specification of Modifications to the Functional Description of the Mission Space Web-based Tool", Master thesis, NPS, June 2001.

Randolph Pugh, "Methods for Determining Object Correspondence during System Integration", Master thesis, NPS, June 2001.

William Windhurst, "An Application of Role-Based Access Control in an Organizational Software Process Knowledge Base", Master thesis, NPS, June 2001.

Report of Inventions: N/A

Technology Transfer:

M. Auguston, V. Berzins, S. Bhattacharya, J. Ge, O. Kiselyov, D. Zhang, members of the Program Committee, the 8th Monterey Workshop

"Engineering Automation for Software Intensive System Integration" (Monterey Workshop 2001), Monterey, California, June 19-21, 2001.

M. Auguston, member of the Program Committee, the 12th IEEE International Workshop on Rapid System Prototyping (RSP2001), Monterey, California, June 25-27, 2001.

M. Auguston, "Visual Meta-Programming Notation", presented at the 8th Monterey Workshop "Engineering Automation for Software Intensive System Integration" (Monterey Workshop 2001), Monterey, California, June 19-21, 2001.

J. Ge, "A Software Agent Framework for Distributed Applications", presented at the 14th International Conference on Parallel and Distributed Computing Systems, Dallas, TX, August 8-10, 2001.

O. Kiselyov, "Subclassing errors, OOP & Practically Checkable Rules to Prevent Them", presented at the 8th Monterey Workshop "Engineering Automation for Software Intensive System Integration" (Monterey Workshop 2001), Monterey, California, June 19-21, 2001.

Luqi, co-chair of the Program Committee, the 8th Monterey Workshop "Engineering Automation for Software Intensive System Integration" (Monterey Workshop 2001), Monterey, California, June 19-21, 2001.

Luqi, General Co-Chair, the 12th IEEE International Workshop on Rapid System Prototyping (RSP2001), Monterey, California, June 25-27, 2001.

J. Michael, "Intelligent Software Decoys", presented at the 8th Monterey Workshop "Engineering Automation for Software Intensive System Integration" (Monterey Workshop 2001), Monterey, California, June 19-21, 2001.

M. Murrah, "Enhancements and Extensions of Formal Models for Risk Assessment in Software Projects", presented at the 8th Monterey Workshop "Engineering Automation for Software Intensive System Integration" (Monterey Workshop 2001), Monterey, California, June 19-21, 2001.

R. Raje, "A Unified Approach for the Integration of Distributed Heterogeneous Software Components", presented at the 8th Monterey Workshop "Engineering Automation for Software Intensive System Integration" (Monterey Workshop 2001), Monterey, California, June 19-21, 2001.

W. Ray "Optimization of Distributed Object-Oriented Servers", presented at the 8th Monterey Workshop "Engineering Automation for Software Intensive System Integration" (Monterey Workshop 2001), Monterey, California, June 19-21, 2001.

M. Saboe, "A Software Technology Transition Engine", presented at the 8th Monterey Workshop "Engineering Automation for Software Intensive System Integration" (Monterey Workshop 2001), Monterey, California, June 19-21, 2001.

M. Shing, Chair, Local Arrangement, the 8th Monterey Workshop "Engineering Automation for Software Intensive System Integration" (Monterey Workshop 2001), Monterey, California, June 19-21, 2001.

M. Shing, Chair, Local Organization, the 12th IEEE International Workshop on Rapid System Prototyping (RSP2001), Monterey, California, June 25-27, 2001.

M. Shing, "The Use of Computer-Aided Prototyping for Reengineering Legacy Software", presented at the 8th Monterey Workshop "Engineering Automation for Software Intensive System Integration" (Monterey Workshop 2001), Monterey, California, June 19-21, 2001.

M. Shing, "DCAPS - Architecture for Distributed Computer Aided Prototyping System", presented at the 12th IEEE International Workshop on Rapid System Prototyping (RSP2001), Monterey, California, June 25-27, 2001.

P. Young, "Use of Object Oriented Model for Interoperability Wrapper-Based Translator for Resolving Representational Differences between Heterogeneous Systems", presented at the 8th Monterey Workshop "Engineering Automation for Software Intensive System Integration" (Monterey Workshop 2001), Monterey, California, June 19-21, 2001.

APPENDICES

FY2001 Publications

Visual Meta-Programming Notation[1]

Mikhail Auguston[2]

Department of Computer Science

Naval Postgraduate School

833 Dyer Road, Monterey, CA 93943 USA

auguston@cs.nps.navy.mil

Abstract

This paper describes a draft of visual notation for meta-programming. The main suggestions of this work include specialized data structures (lists, tuples, trees), data item associations that provide for creation of arbitrary graphs, visualization of data structures and data flows, graphical notation for pattern matching (list, tuple, and tree patterns, graphical notation for context free grammars, streams), encapsulation means for hierarchical rules design, two-dimensional data-flow diagrams for rules , visual control constructs for conditionals and iteration, default mapping rules to reduce real-estate requirements for diagrams, and dynamic data attributes.

Two-dimensional data flow diagrams improve readability of a meta-program. The abstract syntax type definitions for common programming languages and related default mappings (parsing and de-parsing) provide for a practically feasible reuse of those components.

1   Introduction and objectives

Meta-programs are programs manipulating other programs. Typical applications include compilers, interpreters, source code static analyzers and checkers, program generators, and pretty-printers. Domain-specific language implementation and rapidly evolving generative programming [9] are the latest examples of developments in this domain. The complexity and sophistication of meta-programs may be quite significant, so the readability and maintainability become an issue.

Compiler and generator design is a domain that has been studied extensively. There is a pretty good understanding of what to do and how to do it, especially for front-end design, and a lot of domain-specific software design templates are accumulated in literature. The following domain features are among the most common for language processor design.

- Use of context-free grammars to specify syntax and serve as a basis for parser design.

- Intermediate representation of the input in the form of an abstract syntax tree. The importance of different tree data structures is recognized in general for this problem domain.

- Typically, the main components of a language processor are very hierarchical and structured along the structure of data (recursive descent parser is an excellent example of this feature). In other words, language processors are heavily data-based applications.

- It appears that the most commonly used data structures include trees, lists, stacks, tables, and strings.

- The architecture of a language processor in most cases can be represented as a data flow between components (e.g., the famous compiler data flow diagram on the page 13 of the "Dragon Book"[1]).

- The notion of an attribute associated with the data item, and attribute dependency and propagation schemes are of a great relevance (the attribute grammar framework captures some of the essential static checking needs; the data flow analysis performed for the optimization stage in a compiler may be considered as an attribute propagation over the program graph).

---

[2] On leave from New Mexico State University, USA

- Tree (and graph) traversal and transformation is a common template for optimization and code generation tasks.
- Pattern matching (e.g., with respect to regular expressions or context-free grammars) may be a useful control structure for this problem domain.

These considerations and experience with the compiler writing tools RIGAL[2][3], lex and yacc[11], and ELI[10] contributed to this work. Data-flow paradigm is quite natural for meta-programming domain since it is heavily data dependent, and consequently, the graphical notation for data-flow diagrams could be appropriate. This should be integrated with visualization of typical data structures, pattern matching, and encapsulation to provide for well-structured, hierarchical programs. Data-flow diagrams are most commonly used to represent dependencies between data and processes in visual programming languages, for instance, in LabVIEW[5] and Prograph[8].

Two-dimensional diagram notation could significantly improve readability of meta-programs. Some of these ideas have been explored in our previous work[4].

The main suggestions of this work are as follows:

- specialized data structures (lists, tuples, trees),
- data items associations that provide for creation of arbitrary graphs,
- visualization of data structures and data flows,
- graphical notation for pattern matching (list, tuple, and tree patterns; graphical notation for context free grammars and streams),
- encapsulation means for hierarchical rules design,
- two-dimensional data-flow diagrams for rules,
- visual control constructs for conditionals and iteration,
- default mapping rules to reduce screen real-estate requirements for diagrams,
- dynamic (Last #rule $attribute) and static (via associations) data attributes,
- data-flow notation that assumes potential parallelism in the data processing,
- abstract syntax type definitions for common programming languages and related default mappings (parsing and deparsing) that provide for a practically feasible reuse of those components.

2   Constructs

This paper was not intended to give a complete and precise syntax and semantics of the visual language. At this point it is rather a notation that will be upgraded to programming language status after the implementation effort is completed. A (simplified) example of a compiler from a small subset of Lisp (called MicroLisp) to the C language will be used to present the main ideas. Figures 3– 7 present several annotated parsing and code generation rules of the MicroLisp to C compiler. Appendix A contains the MicroLisp context-free grammar and an example of a program.

2.1   Data flow diagrams

Detailed rationale for data-flow diagram notation and a survey of related work can be found in a previous paper[4]. Briefly, a meta-program is rendered as a two-dimensional data flow diagram that visualizes the dependencies between data and processes. Diagrams actually are similar to the notion of procedure in common programming languages. A diagram represents a single function called a rule, and rule calls may be recursive. The data-flow diagram supports the possibility of parallel execution of threads within the rule.

The data-flow paradigm is closely related to the functional programming paradigm [7] and shares with that paradigm referential transparency and good correspondence between the source code (the diagram) and the order of program execution.

Each diagram represents a single function with several inputs and outputs. At the top of a diagram a signature of a rule provides the rule name and types of its inputs and outputs. Besides data items, the diagram may also contain control structures, such as other rule calls, conditional data flow switches, and iterative constructs [4]. All of those constructs are illustrated in the MicroLisp examples.

The rectangular boxes in our notation denote values, and circles and ovals denote patterns, that could be matched with data objects.

2.2 Types

Type represents a set of values (or objects). Basic predefined types include `char` (characters) and `int` (integers). There is also a universal type `ANY` (which is a super type for any type) and the minimal type `NULL` (which is a subtype of any other type and contains a single value `Null` representing also an empty list or tuple).

Aggregate types are ordered tuples of heterogeneous objects, which are useful for abstract syntax representation, and lists (sequences of homogeneous objects that could be dynamically augmented). Extended BNF notation may be used to define tuple types. To a large degree the type system is similar to the type mechanisms in VDM[13] and Refine[12].

Example of a tuple type definition.

```
prog::= function-def* expression
```

This establishes that an object of the type prog is a sequence of zero or more objects of the type function-def followed by an object of the type expression. This could be considered as an abstract syntax representation for the MicroLisp program level. Notice that ordered sequence of objects of the type function-def is nested within an object of the type prog.

Example of a list type definition.

```
text :: [char]
```

There is a predefined list type `id:: [char]`, which stands for a set of character strings that are valid identifiers.

Example of a type definition with several alternatives (union type).

```
expr :: int | id | simple-expression
```

This effectively declares that types int and id are subtypes of expr in the scope of this definition.

Appendix B presents some of the type definitions for the MicroLisp example.

2.3 Default mappings
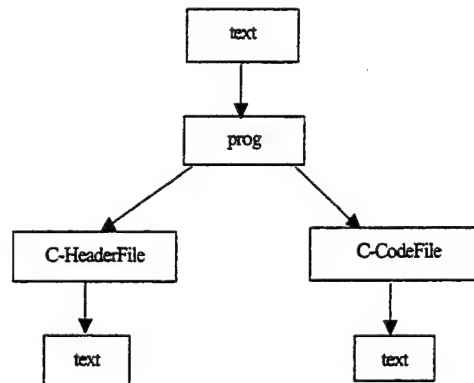
Figure 1. The top level data flow diagram for MicroLisp to C compiler

Certain rules may be declared as default mappings. It means that corresponding rule calls are optional in the diagrams, and input and output data boxes may be connected directly. This helps to save some screen real estate and to make diagrams less crowded and more readable. Typically default mappings may be introduced for text-to-abstract syntax (parsing) and for abstract syntax-to-text mappings (de-parsing, or abstract syntax-to-concrete syntax mappings).

Yet another kind of default mappings is associated with concatenation operations for tuples and sequences. In fact this is a composition of parsing and de-parsing default mappings applied in the context of (visualized) concatenation. See MicroLisp generation rules for examples (Figures 6-7).

Definitions of abstract syntax types for common programming languages and related parsing and de-parsing default mappings may be valuable assets for reuse.

Default mappings also open the road for "lightweight" inference. For example, suppose that type A is defined as follows:

```
A :: B | C
```

and there are default mappings B -> D and C -> D, then it is possible to derive a default mapping for A -> D. This example actually addresses the polymorphism issue in our lightweight type system. Similar inference rules could be developed for other aspects of type system based on transitivity of subtype relation.

2.4 Associations

Data objects may be associated with other data objects. Each of those objects may have other associations as well. Associations are not a necessary part of the type definition (although they could be included in the type definition as well) and are rather optional named attributes of particular objects. Associations may be used to create arbitrary graphs from objects. The following picture on Figure 2 illustrates the creation of a graph structure via associations from three data objects. Association is not symmetric. According to the following diagram object A has been associated with an attribute B via an association named ab, object B with C via bc, and C with A via ca.

Associated objects are retained when the host objects are the source and target in an identical transformation (plain arrow connecting data boxes of the same type) or are passed as inputs and outputs of rule calls. A special built-in rule #COPY creates a copy of an object but retains only those components declared in the type definition. Associated objects could be retrieved by pattern matching. For instance, on the right-hand diagram on Figure 2, object C (belonging to the associations established in the previous example) may be passed as input, and an access to objects B and A can be obtained via pattern matching (circles denote object patterns here). Notice that the direction of association arrow indicates the access path from the host object to the attribute object. The association mechanism may be useful to simulate attribute-grammar–like attribute propagation in ensembles of objects, to represent collections of objects as graphs, to implement symbol tables (where identifiers may be represented as associations names), and so on.
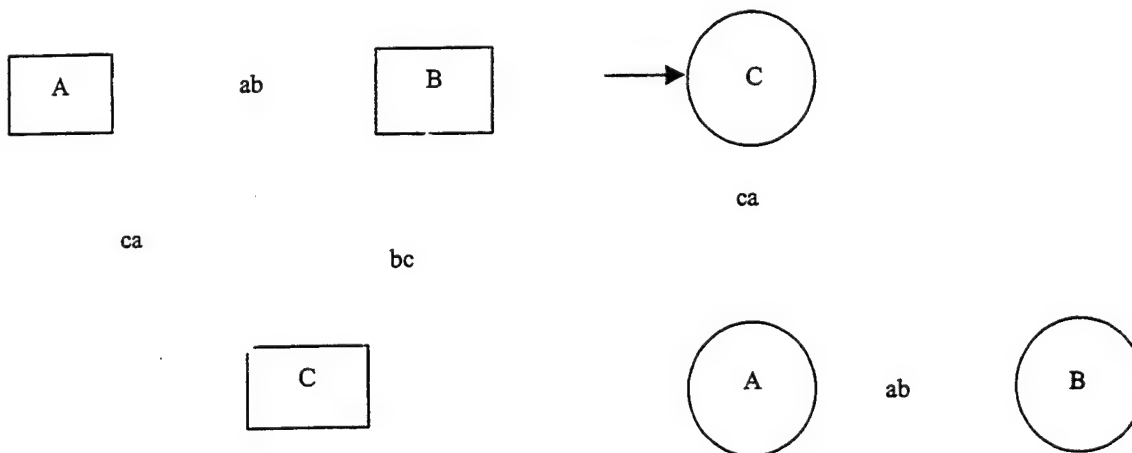
Figure 2. Construction of associations between objects and retrieval of them
using pattern matching

2.5 Patterns and streams

Data object patterns are used to visualize structure of objects in order to provide access to object components and associated objects. An object pattern may be placed in any part of the data flow and is matched with the object connected to the pattern input.

11

If pattern matching is successful the input object is passed downstream. If pattern matching fails, the entire diagram execution fails, and the diagram sends to its outputs a default value Null, unless the pattern has been provided with the 'Failed' output route. See MicroLisp rules in Figures 3-4 for examples.

If a rule's input is a list, patterns applied to this input may be chained in a sequence (using thick gray arrows) to be applied consecutively. This pattern sequence consumes as many objects from the stream as it can successfully match. The notion of stream corresponds to the sequence in RIGAL language[2][3], and semantics of pattern matching is derived from RIGAL's pattern matching semantics. See MicroLisp parsing rules for example (Figures 3-5).

Rules can create output streams of objects as well.

2.6 States and dynamic attributes

Rule may have states – objects that persist while rule instance is active and can be updated by assignment operators within the rule or from other rules called from this rule. This mechanism could be actually considered a macro extension for diagram notation when a corresponding state object is passed to the called rules as an additional parameter and returned back to the callee as an additional output. States have names starting with the $ symbol, e.g. $X. The reference to the rule's #A state $X has a form Last #A $X. When referred within the rule #A, the prefix Last #A can be dropped. See Figures 4-5 for examples.

3 Examples of MicroLisp to C compiler rules

The following diagrams present three top level parsing rules and two top level generation rules for MicroLisp -> C compiler. They illustrate most of the notations discussed above. Additional annotations provide more specific details and discussion. Those rules are deployed according to the data flow diagram on Figure 1 and default mappings in Appendix B.

3.1 Parsing

The source code of MicroLisp program is represented as a stream of characters. It is assumed that there is a lexical component that filters out comments, spaces, tabs, end-of-line characters from the stream before it is fed to the parsing rules.

```
#program: Stream [char]-> prog, Stream [message]
    state $func-list: [id]   -- updated by #func-def
```

#func-def

'?'

#expr

***

Failed

message:
"? expected"

Syntax err

$func-list

Func-tab

Syntax err

message:
"Errors
detected"

prog:

func-def ... expr
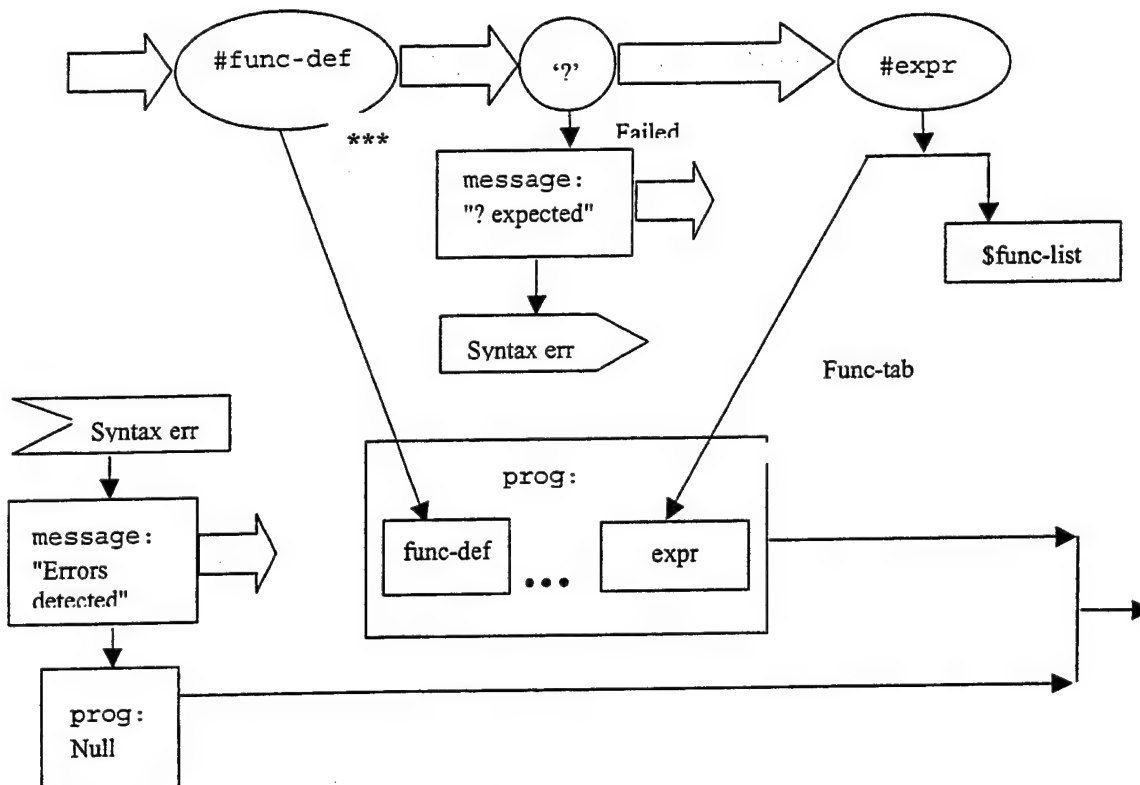
prog:
Null

Figure 3. Parsing rule for the grammar rule
```
        program ::= func-def * '?' expression
```

Annotations for the rule #program

- This rule has a state $func-list which will be gradually updated by the rule #func-def calls (see Figure 4). At the end of parsing, object $func-list will be added as an attribute (via association with the name Func-tab) to the resulting object of the type prog. The box containing $func-list has a dummy input of the type ANY, which is activated when the last pattern #expr terminates with success. This ensures the timing when the state value is picked up for the association operation.

- The rules #func-def and #expr are used as patterns. If pattern matching encapsulated in these rules is successful, the rules also are successful and return values, which are used to assemble the return value of the rule #program.

- If pattern matching for the pattern '?' fails, the entire rule #program also fails and returns object Null, but before it happens two messages will be sent to the output stream. Markers labeled 'Syntax err' are used to prevent a mess with arrow intersections.

- A data flow fork denotes duplication of the data item sent to two or more threads.

- Nesting boxes and forwarding output of pattern rules of the types func-def and expr inside the resulting box of the type prog provide an intuitive visualization for the tuple constructor.

- The application of pattern #func-def may be repeated zero or more times (indicated by the ellipsis '***'), and it is synchronized with the tuple constructor (as the box of the type func-def in the resulting prog box is also accompanied by an ellipsis).

```
#func-def: Stream [char]-> Func-def, Stream [message]
           state $param-list: [id]    -- used in #expr
```
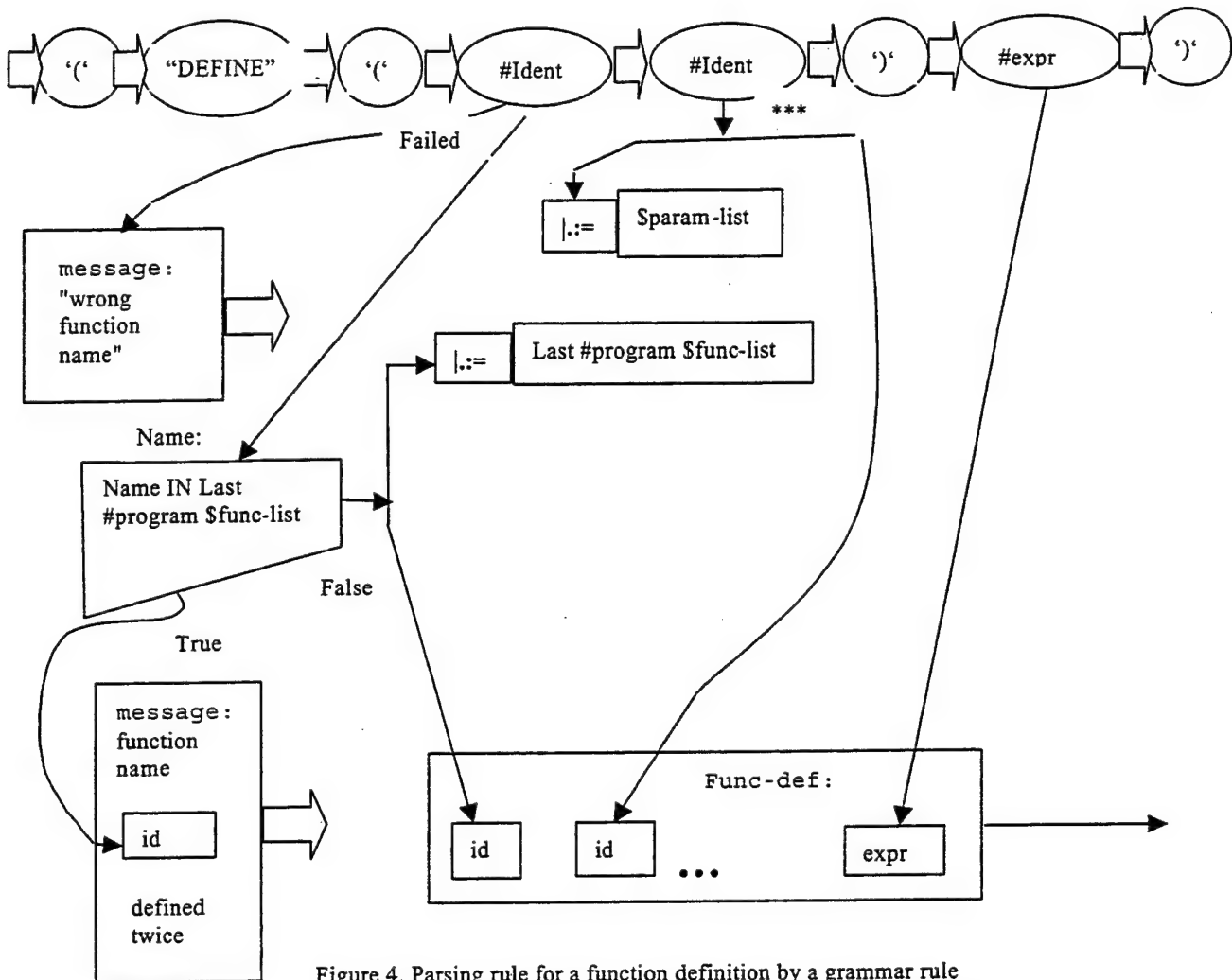

Figure 4. Parsing rule for a function definition by a grammar rule
Function-definition ::= '(' DEFINE '(' Name Param * ')' Expression ')'

Annotations for the rule #func-def

- Built-in rule #Ident matches a character string that is an identifier. When successful, this identifier (an object of the type id) is input to the conditional data flow switch to check whether the function name is already on the list. If true, the id item is forwarded to the message output stream. If false, it goes to the resulting tuple constructor.

- A function name is also sent to update state $func-list in the current instance of rule #program. |.:= stands for the operation to append an element to the end of list. This assignment operation updates the state Last #program $func-list.

- The entire sequence of patterns in this rule consumes part of the input stream delegated from the calling rule #program.

- Parameter names are appended to the state variable $param-list. All state variables are initialized by Null, which stands for empty list in this case.

14

```
#expr : Stream [char] -> expr, Stream [message]
```
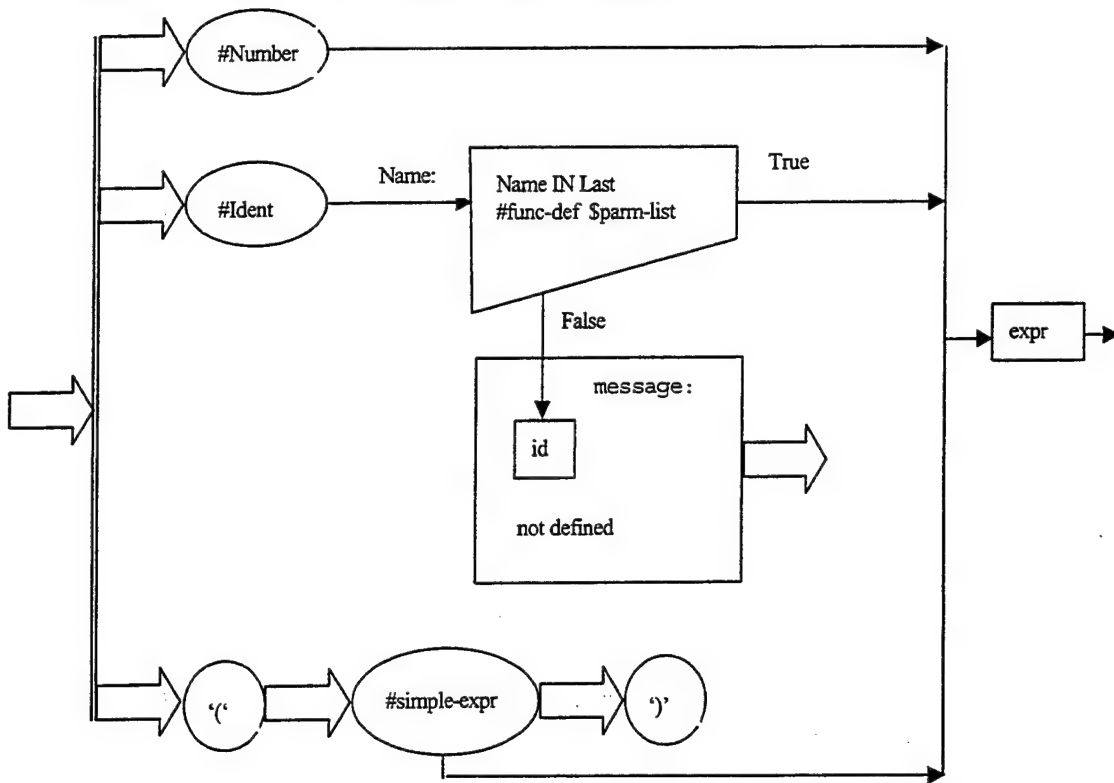
Figure 5. Parsing rule for MicroLisp expression for the grammar rule
```
expression::= integer | parameter-name | '(' SimpleExpression ')'
```

Annotations for the rule #expr

- A pattern may have several alternatives. The alternatives are applied in order of appearance, if the first alternative fails, the pattern matching backtracks in the input stream and the next alternative is applied until one of alternatives is successful. If all alternatives fail, the entire alternative pattern also fails.

- The built-in rules #Number and #Ident, when successful, return objects of the types int and id, correspondingly. Since the type expr is defined as a supertype for int and id, the data flow to the resulting object of the type expr is consistent.

3.2   Code generation

Code generation rules take as input a MicroLisp abstract syntax object and output C abstract syntax objects. Target code template representation in the diagrams is based on default mappings for C abstract and concrete syntax and visual representation of append operation as nested boxes.

Annotations for the rule #gen-program

15

```
#gen-program: prog -> C-HeaderFile, C-CodeFile
```
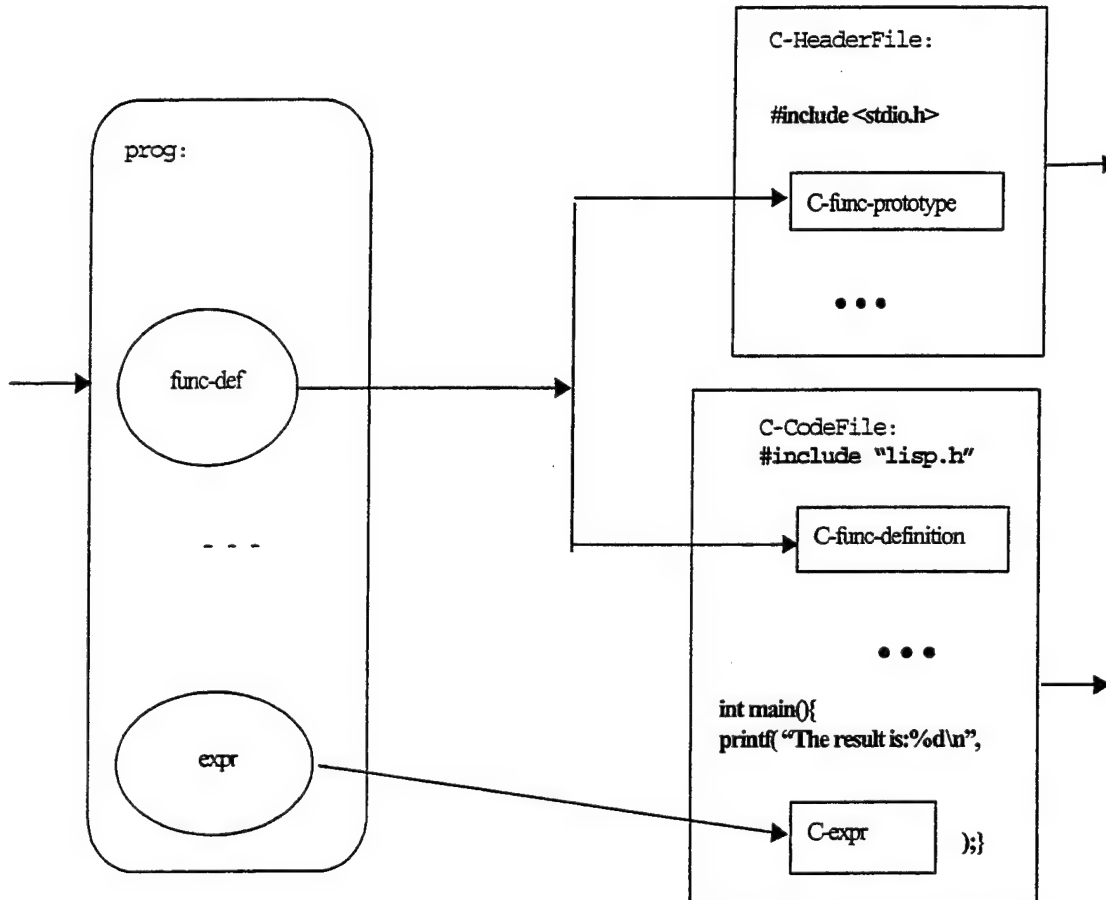

Figure 6. Generation rule for the MicroLisp program level

- The input is of the type prog (abstract syntax object for MicroLisp) and a pattern for this object provides an access to the component retrieval. Since func-def components may be repeated zero or more times, the ellipsis in the pattern represents the iterative traversal.

- The iteration of the input is synchronized with the iterative generation of objects in two outputs. The transformations itself are carried by default mappings `func-def -> C-func-prototype` and `func-def -> C-func-definition`. The rule #gen-function-prototype in the next example gives the algorithm for the first of these default mappings. Since the template provides particular concrete syntax for parts of the C code, those text segments will be stored with corresponding C abstract syntax objects. The resulting parse tree for include and printf will contain objects of the type id and text-string that hold values, such as "int", "printf", and other. These concrete syntax values are retrieved by default mappings when pretty-printing corresponding C abstract objects.

- The rule #gen-program constructs the target C code in the abstract syntax form. The mapping from abstract syntax to the text will be done according to the main diagram in Figure 1 by corresponding de-parsing default mappings for the C language. Both the abstract syntax definitions and default parsing and de-parsing mappings for the C language may be reused for any other meta-program that uses C as a target.

16

Annotations for the rule #gen-function-prototype

- This rule provides the flavor of hierarchical structure of generation templates.
- The first appearance of the string "int" in the target object C-func-prototype object will be converted by the C default parsing mapping into object C-type and the string "int" will be associated with it as a value. The same is true also for the iteration of "int" in the parameter list.
- Box around the second instance of "int" is needed to indicate the binding with the iteration of id in the source object func-def.

```
#gen-function-prototype: func-def -> C-func-prototype
```
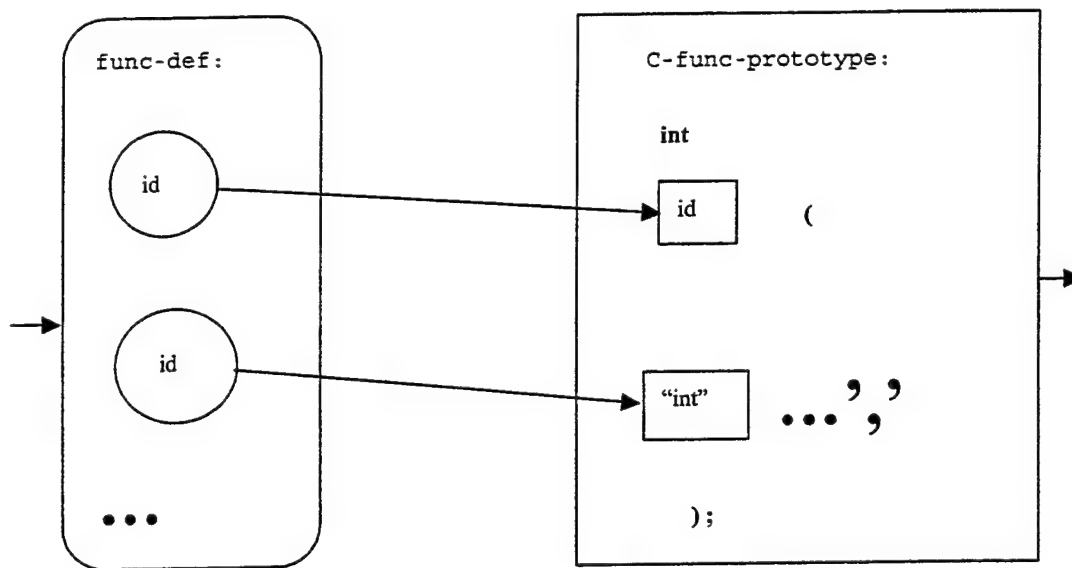

Figure 7. Generation rule for C function prototype.

- Parentheses, semicolon, and comma (as a separator between iterated elements; in the graphical interface there should be a way to indicate that comma is related to the iteration ellipsis) in the target object are optional, and if present, will be consumed by corresponding C default parsing mappings. The resulting object is still an abstract syntax object.

4   Preliminary conclusions

This paper presents very preliminary results on the visual notation for meta-programming. Work continues on the language itself, case studies, and implementation issues. At the moment of this writing the interpreter for the core of data-flow language is already implemented, and work is in progress on the graphical editor and advanced features like default mappings and tuple pattern matching. In it current form, the concepts presented may be used as a useful supplement to the meta-program design documentation. We expect the advantages of this approach to be as follows.

- Visualization of data and data flow provides for better readability and uncovers parallelism in data processing.
- The tuple type provides for a precise, disciplined, and flexible way to define abstract syntax.
- The simple association mechanism provides a natural way to introduce data attributes and opens the road for processing of arbitrary graphs without cluttering the language with additional means.
- Pattern matching notation covers in a uniform way data objects, rule calls, associations, and extended BNF notation for parsing.

- The language provides for systematic and consistent correspondence between constructors and patterns.
- The dynamic attributes (states) are actually macro extensions of pure functional paradigm (may be considered as additional inputs and outputs for diagrams referring to the states), provide for more efficiency, and make the data flow diagram simpler and less cluttered.
- Default mappings may be very convenient for generation templates, provide basis for lightweight type inference, and rule reuse.
- Data streams and patterns give a flexible and expressive framework for parsing rules supporting extended BNF notation, support reasonable and informative parsing error messages.
- Control mechanism, such as data flow switch, iteration and recursion fit well with data-flow notation and provide for transparent and expressive language to define different kinds of meta-programming algorithms.

References

[1] A.Aho, R.Sethi, J.Ullman, Compilers: Principles, Techniques, and Tools, Addison-Wesley, 1986

[2] M.Auguston, "RIGAL - a programming language for compiler writing", Lecture Notes in Computer Science, Springer Verlag, vol.502, 1991, pp.529-564.

[3] M.Auguston, "Programming language RIGAL as a compiler writing tool", ACM SIGPLAN Notices, December 1990, vol.25, #12, pp.61-69

[4] M.Auguston, A.Delgado, Iterative Constructs in the Visual Data Flow Language, in Proceedings of IEEE Symposium on Visual Languages, Capri, Italy, 1997, pp.152-159

[5] E.Baroth, C.Hartsough, Visual Programming in the Real World, in Visual Object-Oriented Programming, Concepts and Environments, (ed. M.Burnett, A.Goldberg, T.Lewis), Manning 1995, pp.21-42

[6] D.Batory, Gang Chen, E.Robertson, Tao Wang, Design Wizards and Visual programming Environments for GenVoca Generators, IEEE Transactions on Software Engineering, Vol. 26, No 5, May 2000, pp.441-452

[7] R. Bird, T. Scruggs, M. Mastropieri ,Introduction to Functional Programming, Prentice Hall, 1998

[8] P.T.Cox, F.R.Gilles, T. Pietrzykowski, "Prograph", in Visual Object-Oriented Programming, Concepts and Environments, (ed. M.Burnett, A.Goldberg, T.Lewis), Manning 1995, pp.45-66

[9] K.Czarnecki, U.Eisenecker, Generative Programming, Methods, Tools, and Applications, Addison Wesley, 2000, pp.832, ISBN 0-201-30977-7

[10] R. W. Gray, V. P. Heuring, S. P. Levi, A. M. Sloane, and W. M. Waite. Eli: A Complete, Flexible Compiler Construction System, Communications of the ACM, 35(2):121-131, February 1992.

[11] J. Levine, T.Mason & D.Brown, lex & yacc, 2nd Edition, O'Reilly, 1992

[12] Reasoning Systems, "Refine User's Guide", Palo Alto, 1992

[13] The Vienna Development Method: The Meta-Language, D. Bjorner et al, eds, LNCS 61, Springer 1978

Appendix A. Syntax of MicroLisp language and an example of a program

```
Program ::= Function-definition* '?' Goal-Expression

Goal-Expression ::= Expression

Function-definition ::= '('DEFINE'('Function-name Parameter-name*')' Expression ')'

Expression ::= Integer | Parameter-name |'(' SimpleExpression ')'

SimpleExpression ::= BinOperation Expression Expression | UnOperation Expression |
                 Function-name  Expression* |COND Branch + | READ_NUMBER

Branch::= '('Expression Expression ')'

BinOperation ::= ADD | SUB | MULT | DIV | MOD | EQ | LT |GT | AND | OR
```

```
UnOperation ::=  MINUS |  NOT
Function-name  ::= Identifier
Parameter-name   ::= Identifier
```

Example of a MicroLISP program.

```
( DEFINE ( gcd  x y)
   (COND (EQ x y)  x )
  ( (GT x y) ( gcd  (SUB x y)  y ) )
  (  1 ( gcd  x (SUB y x)  ) ) )  )
?  (gcd  (READ_NUMBER)  (READ_NUMBER) )
```

Appendix B. Type definitions for MicroLisp -> C compiler

```
message::  [ char ]
program:: ( func_def* expr)| NULL
         attribute func_tab: [id]
func_def:: id id* expr
expr:: number | id |(op expr expr)|(op expr)|read_num | cond | function-call
function-call:: id  expr*
cond:: (expr expr)*
```

default mappings

```
 #prog: [ char ] -> prog
 #gen_program: prog -> C-HeaderFile, C-CodeFile
 #gen-function-prototype: Func-def -> C-func-prototype
 #gen-function-def: Func-def -> C-func-definition
 #pretty_print_prog: prog -> [ char ]
```

This is a sketch of a (over)simplified version of C abstract syntax.

```
 C_CodeFile::  include-statement * C-func-definition +
 C_HeaderFile:: include-statement C-func-prototype *
 C_func_prototype:: C-type func-name C-type *
 C-type:: id
 C_func_definition:: ..........
 C_expr:: .....
```

Default mappings include parsing rules and pretty-printing rules (abstract syntax to text mappings).

19

A Software Agent Framework for Distributed Applications[+]

Jun Ge, Boon Kwang Kin, Valdis Berzins
Department of Computer Science
Naval Postgraduate School
Monterey, CA 93943, USA
Email: {gejun, berzins}@cs.nps.navy.mil, bkkin@nps.navy.mil

Abstract

Software wrapper and glue technology is used to build the architecture for distributed systems. This paper proposes a simple framework using agents to act as interfaces among processes interacting and cooperating to support the wrapper-glue architecture. These agents encapsulate the implementation details and make the network transparent to the running processes. The proposed framework is built on JINI infrastructure and uses Linda TupleSpace type model of communication mechanism for processes to interact with one another. The agent interface is written in Java programming language with two language wrappers, C Library wrapper and ActiveX Component wrapper to support processes written in multiple languages including Java, C++/C, Ada and Visual Basic. The agents can run on platforms with JVM support. This agent framework serves in the development of distributed systems as the "glue" among components for communications. Test examples implemented in various languages are provided.

Key words: wrapper and glue, agent, distributed system, JINI, software wrapper

1. Introduction

In the last few years, the computing landscape has changed dramatically, as more devices such as hand phones, PDAs (Personal Device Assistance) and internet terminals, become network-connected, and as more companies depend on the Internet to operate and communicate; distributed applications (one that involves multiple processes and devices) will become the natural way we build systems, while the standalone desktop applications will become out-dated and less commonly built.

Distributed applications offers many benefits compare to standalone applications such as gain in performance, better scalability, resource sharing, fault tolerance and availability. Despite their benefits, distributed applications are difficult to design, build and debug. The distributed environment introduces many problems that are not concerns when writing standalone applications.

Some of these problems are heterogeneity, latency, partial failure, synchronous and coordination.

Rewriting legacy software to run in a distributed environment tends to be prohibitively expensive and complex. Many of this legacy software are expensive investment developed over many years, replacing them with new designs is usually not easily justifiable in term of cost and resource allocation. Although, the only way to keep such legacy software useful is to incorporate them into a wider cooperating community in which they can be exploited by other pieces of software, this tends to be very complex in software design.

Recently, the techniques to "glue" multiple processes running in a heterogeneous environment range from low level sockets and messaging techniques to more sophisticated technologies object resource broker (CORBA, DCOM). Many of these techniques either require developers to perform significant work in constructing the communication mechanisms or need developers to have a good knowledge of the interface details before designing. Hence, "glue" pieces of processes are a difficult task and require skillful designers.

Existing technologies for distributed system design include these models, namely client/server model and distributed object model.

The client/server model contains a set of server processes; each one acting as a resource manger for a collection of resources of a given type such as database server, file server, print server. All shared resources are held and managed by the server processes. Beside server processes, it also contains a collection of client process; each one performs a task that requires access to some shared hardware and software resources. The client/server model is a form of distributed computing in which the client communicates with the server for the purpose of exchanging or retrieving information. Both the client and server usually speak the same language (protocol) to communicate. The major problem with client/server model is that the control of individual resource is centralized at the server; this could create a potential

bottleneck and a single point of failure. Moreover, to improve performance and cater to increasing number of clients, implementations of similar functions are usually replicated multiple servers. On the other hands, the centralized of resources at a single location greatly simplifies the management of these resources. The client/server model can be implemented in various ways. Typically, it is done using low-level sockets, remote procedure calls or high-level message oriented middleware such as message queue.

A distributed object-based system is a collection of objects that isolates the requestor of services from the providers of services (servers) by a well-defined encapsulating interface. Clients are isolated from the implementation of services as data representations and executable code. In distributed object model, a client sends a message to an object, which in turn interprets the message to decide what service to perform. This service could be performed either through the object or a broker. Distributed object systems such as CORBA, DCOM, and Java RMI provide the infrastructure for supporting remote object activation and remote method invocation in a client-transparent way. A client program obtains a pointer (or a reference) to a remote object, and invokes methods through that pointer as if the object resides in the client's own address space. The infrastructure takes care of all the low-level issues such as packing the data in a standard format for heterogeneous environments (i.e., marshaling and unmarshaling), maintaining the communication endpoints for message sending and receiving, and dispatching each method invocation to the target object. Among all the different vendors for distributed object systems, CORBA is the most widely supported standards. Its advantages are platform independence, open industry standards that contains over 750 industry members.

Jini is one of a large number of distributed systems architectures, including industry-pervasive system such as CORBA and DCOM. It is distinguished by being based on Java programming language, and deriving many features that leverage on the capabilities that this language provides, like object-oriented programming, code portability, RMI (Remote Method Invocation), network support and security.

Some of the features Jini Technologies offers include enabling users to share services and resources over a network, providing easy access to resources anywhere on the network while allow the network location of the user to change, and simplifying the task of building, maintaining, and altering a network of devices, software, and users.

Jini technology consists of a programming model and a runtime infrastructure. The programming model helps designer build reliable distributed systems, as a federation of services and client programs. The runtime infrastructure resides on the network and provides mechanisms for adding, subtracting, locating, and accessing services as the system is used. Services use the runtime infrastructure to make themselves available when they join the network. A client uses the runtime infrastructure to locate and contact desired services. Once the services have been contacted, the client can use the programming model to enlist the help of the services in achieving the client's goals.

Tuple Space model was first conceived in the mid-1980 at Yale University by professor David Gelernter[1] under a project called Linda. Tuples are typed data structures. Collections of tuple exist in a shared repository called a tuple space. Coordination is achieved through communication taking place in a tuple space globally shared among several processes; each process can access the tuple space by inserting, reading or withdrawing tuples.

In this model, the programmer never has to be concerned with or program explicit message passing constructs and never has to manage the relatively rigid, point-to-point process topology induced by message passing. In contrast, coordination in Linda is *uncoupled* and *anonymous*. The first means that the acts of sending (producing) and receiving (consuming) data are independent (akin to buffered message passing). The second means that process identities are unimportant and, in particular, there is no need to ``hard wire'' them into the code.

Software wrapping is a technique in which an interface is created around an existing piece of software, providing a new view of the software to external systems, objects, or users. Wrapping can be accomplished at multiple levels: around data, individual modules, subsystems, or entire systems. There is not standard specifically for wrappers. Wrappers can be used to interface legacy code to standardized architectures. For example, IDL is implemented via tools that automatically generate wrappers to interface to CORBA.

This paper tries to integrate the effort on both JINI technology and software wrapping method in rapid prototyping practice [2]. A simple framework is proposed by using software agents to act as interfaces among various processes that interact and cooperate in distributed environment. It shields developers from the underlying dynamic and complex network environment, offers developers a simple set of APIs (Application Program Interface) to build distributed applications without worry about their platform and programming languages, and presents exiting software a easier way to interact and cooperate with other applications in heterogeneous environment. Therefore, the proposed agent framework becomes a concrete implementation for

glue library in wrapper and glue architecture. Session II presents an overview of the proposed framework and a simple description of its features and underlying design. The language wrapper design for the framework is introduced in Session III. Session IV gives a test-bed example of using the agent frame in multiple language wrappers.

2. Design

An overview of our framework is given in this session. It also describes the underlying design and the features of our agents.

The proposed framework builds on JINI infrastructure and uses JINI network technology to simplify the task of building and maintaining reliable distributed systems (Figure 1). This technology consists of a well-defined programming model, allowing us to easily create our own service and leverage on services already built to support JINI infrastructure. Using this programming model, we do

not have worry about the low-level communication protocol. Client processes can dynamically locate and access services held in the JINI community using its runtime infrastructure, even if they do not know their host URL addresses.

The framework uses a Linda TupleSpace model type of communication mechanism for inter-processes communications. Processes are loosely coupled, rather than through direct communication, they interact in a globally shared space - *repository service* (provides by JavaSpace Service), through share variables - *entries*. Being loosely coupled, processes need not be physically connected all the time and do not have to worry about the point-to-point topology induced by message passing. Several processes residing on same machines or on different machines can access the repository simultaneously. They interact among each other by means of reading, writing or consuming entries stored in the repository service.
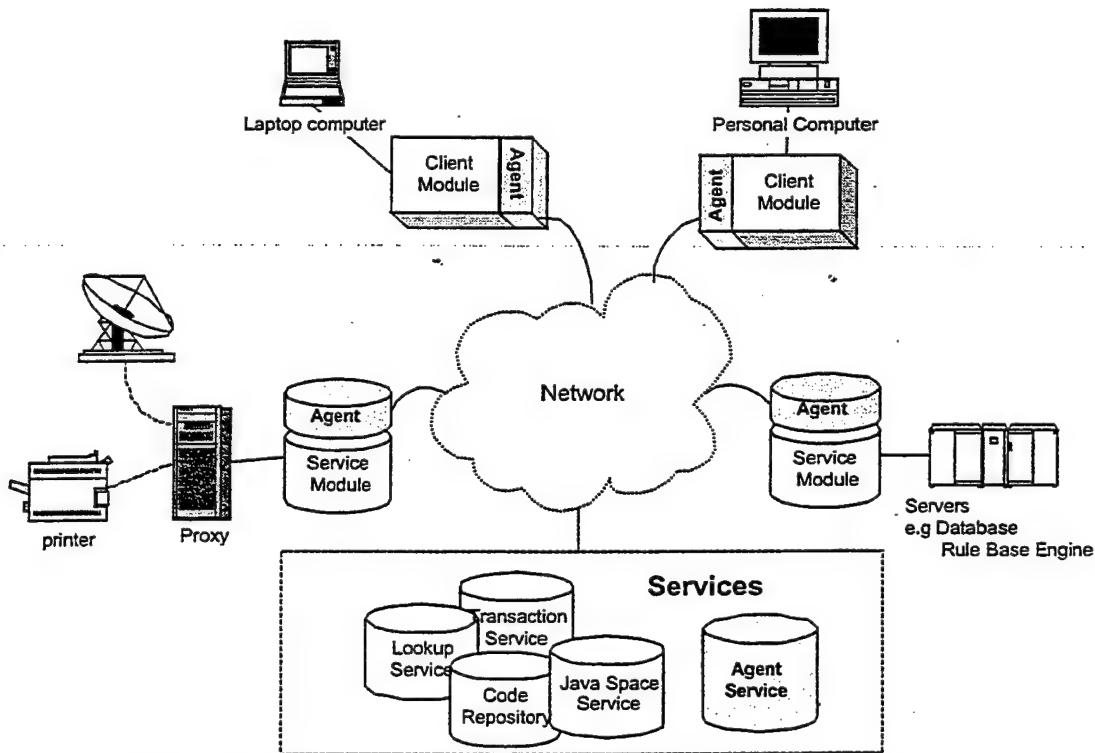
Figure 1. An example of a distributed application using the agent framework

Repository service is a shared, network-accessible depot for entries storage. It behaves like a lightweight relational database, where agents acting on behalf of their processes can store, retrieve and query entries stored in it. Unlike database, where users construct Structured Query Language (SQL) statements to query records; agents use pre-constructed templates, defined in our framework, to

match entries store in the repository; only entries that match exactly the data types and fields defined in the template are returned by the repository service.

Entries are collection of values or objects place in repository service by coordinating processes for information sharing. Before a process can start operating

on an entry, it first has to declare the entry, identifying by a unique name and an entry type, with its agent; just like variables declaration in programming techniques. The entry type varies from simple primitive types (like integer, float, double and etc) to more complex types (like queue, stack, list and etc), where process can manage entries as groups. The entry, upon declaration, is assigned an entry handler to serve operations for accessing the repository service.

Entry handler is responsible for carry out operations pertaining to a declared entry. There are many kinds of entry handlers, each one is associated to an entry type and has methods designed specifically to handle that particular entry type. Methods that are common in all handlers are: read, write, take, update and notify methods, process mainly use them for manipulating entries store in repository service. Each entry handler consists of a set of attributes that determine how it carries out its operations. Many of them can be overwritten, after entry declaration, by processes to meet different application needs. For instance, an entry-leasing attribute, which determine the validity of the entry process store in repository, can be used a real-time application to specify the deadline of information, preventing the receipts from accessing obsolete information, which sometime can be more damaging than not have any.

Establishing a session with agent service is done in two simple steps: first locate the service and then perform a login registration. If process knows the network address where agent service is located, process can bypass the search procedures. Searching for services in JINI network is done using multicast protocol - agent inserts a package into a network and wait for lookup services to respond, a lookup service is a facility where services publish their services. The lookup services, upon receiving the request package, response by returning a list of service items, each item describes its service properties and functions. Agent search through the list, comparing their service attributes with those of the agent service. After it has determined a match, it proceeds to establishing a connection follow by service registration, providing a valid login ID and a password to the agent service.

Below is a summary of features the framework provides,
  ✓ A simple and yet comprehensive interface that allow multiple processes to get connected and interact with one another in a distributed environment.
  ✓ Processes can be written in Java, Visual Basic, C/C++, or Ada; two agent wrappers are included, ActiveX wrapper and a C library wrapper.
  ✓ Processes are loosely coupled; they need not be physically connected all the time and do not have

to worry about the point-to-point topology induced by message passing.
  ✓ Several processes residing on same or different machines can access the repository service and retrieve data simultaneously in a reliable manner.
  ✓ Agent Service provides authentication and control mechanism to manage processes using its services.
  ✓ Avoid the needs to create and manage remote/virtual classes (e.g. stubs and skeletons in RMI and CORBA implementations)
  ✓ Provide callback mechanism that invokes user-defined methods when conditions are met.
  ✓ Support transaction, enforcing consistency over a set of entry operations
  ✓ Support leasing, preventing resources from growing out of bound.

3. Language Wrappers

There are many compelling reasons for the agent to support a wider variety of programming languages beside Java language. Some of these reasons are software reuse, integration with legacy code, leveraging on tools that are not available for Java language, and performing low low-level activities such as hardware interface.

Two agent wrappers, ActiveX Component Wrapper and Native C Library Wrapper, are implemented. ActiveX Component Wrapper allows our agent to be encapsulated as objects in Visual Basic, Visual C++ or Microsoft Office applications running in Window platform, whereas Native C Library Wrapper allows our agent to be bind together with native languages such as Ada, C and C++.

The wrapper modules consist of two separate components: an ActiveX wrapper and C Library wrapper (Figure 2). An ActiveX wrapper embedded the agent as object such that it can be call by process written in Visual Basic, Visual C++ or Microsoft Office application in window platform environment. C Library wrapper allows the agent to be bound together with processes written using machine dependents languages like C, C++ or Ada.

The implementation of the ActiveX Wrapper was done using a packager, an ActiveX Packager for Java Bean, that come along with JVM plug-in provided by Sun MicroSystem. This packager automatically generates the wrapper for any Java bean by going through sequence of pre-compiling. Two files are eventually produced after the process, an OCX (OLE Control Extension) and TLB (Type LiBrary). To make the OCX available to the window environment, developers have to explicitly register them with the window registry.
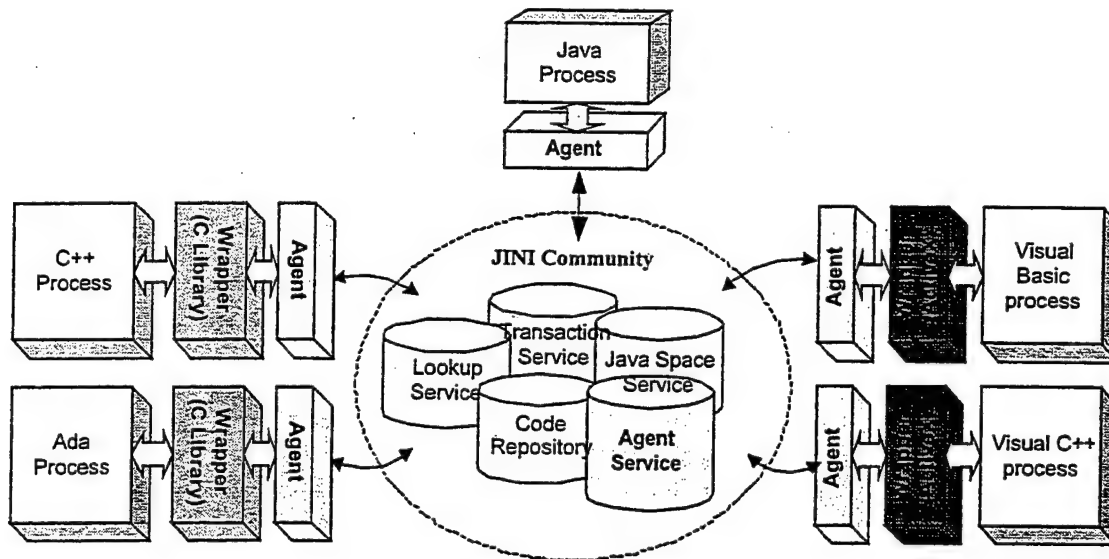
Figure 2. Agent wrappers

Together with the Java Bean Bridge and JVM (Java Runtime Environment), any method calls on this OCX component will marshaled over the bridge and gets executed in the JRE memory space; the return for the function is unmarshalled by the bridge and given back to the OCX component.

··The C Library Wrapper was build using JNI (Java Native Interface) APIs and the building process is more complicated and tedious compare to implementing the ActiveX Wrapper. We have to map every Java types to C use in agent interfaces, create corresponding interfaces in C language for every methods defined in the agent interfaces, and manage the memory resources to prevent memory leak.

4. Example of Language Wrappers

Three simple test programs are created, written in a different language, to test the configuration of services and client processes. These three programs serve as a distributed system to share information. Figure 3a to 3c show the graphical user interfaces of these test programs: a Java GUI, Visual Basic GUI and C GUI respectively. These test programs has implemented most of the commonly used functions descript in our framework. Besides, using them for testing the setup, they also provide another source for developers to understand how some of the features descript in our framework works.

Once the JINI services (includes our agent service) are started, run the test programs on separate machines, these machines must share a common network. Next, update the agent setting, by overwriting the fields under "Agent Setting" header, if the setting varies from our defaults. Following that, press the "initAgents" button, it will show

a message "agent connected" if it successful with agent service. Declare a new entry, with same entry name and of same type, on both machine using those buttons locate on the lower left-hand panel.
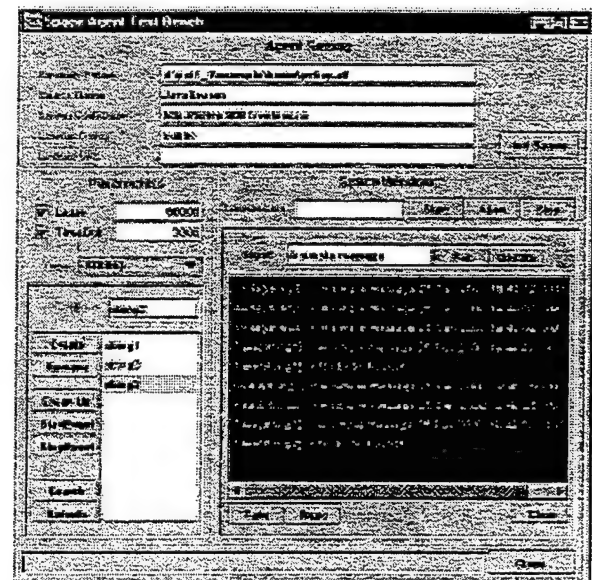
Figure 3a. Java Language version Test Bench
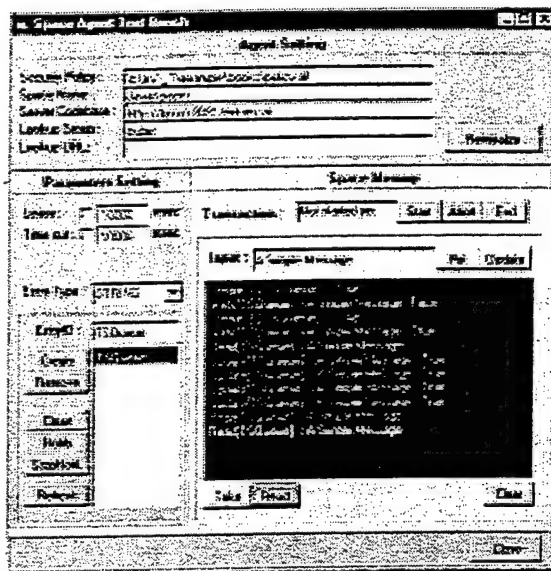
Figure 3b. Visual Basic Language version Test Bench
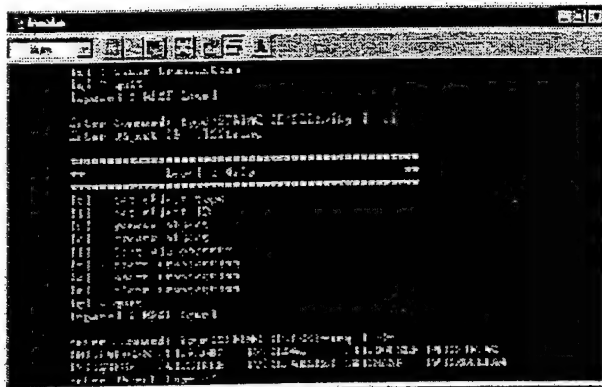
Figure 3c. C Language version Test Bench

5. Conclusions

The proposed agent framework with the language wrappers focuses on the high-level architectures and process interfaces. It ensures that the design is scalable and process can be written in a variety of programming languages by designing a common set APIs and building wrappers for non-Java processes. The framework can be used in various domains. The implementation has been chosen to build on top of JINI/JavaSpace. It provides the following benefits for distributed system design.

(1) It avoids the need to create or to manage the remote/virtual classes (e.g. stubs and skeletons in RMI and CORBA implementations)
(2) The development of dynamic and distributed applications is relatively easy. Programmers do not have to manipulate remote accesses to objects/classes (such as the stubs and skeletons in Sun's RMI or the virtual objects.

(3) It eliminates the process of encoding and decoding process for the case of message passing.
(4) It is easy to integrate with existing software components.
(4) It provides good scalability. Currently two services, transaction service and repository service, are used, while new services can easily be added, such as database connection/query.
(5) It enables a callback mechanism that invokes user-defined methods when conditions are satisfied.

With these advantages, the proposed agent framework plays as the communication layer glue in the software wrapper and glue technology. It provides a concrete implementation practice for glue library for higher-level wrapper to call. The whole architecture is used in rapid prototyping architecture for heterogeneous distributed systems.

References

[1] D.Gelernter, *Generative Communication in Linda*, ACM Trans. Programming Languages and Systems, 7(1), Jan.1985, pp. 80-112

[2] Luqi, "*Computer-aided prototyping for command and control system using CAPS*", IEEE Software, 9(1), Jan. 1992, pp 56-67

[3] Cloudscape, *Cloudscape Java database*, http://www.cloudscape.com

[4] Bill Joy, *The Jini Specifications*, Addison Wesley, Inc., 1999

25

JAVA Wrappers for Automated Interoperability

Ngom Cheng, Valdis Berzins, Luqi, and Swapan Bhattacharya

Department of Computer Science
Naval Postgraduate School
Monterey, CA. 93943 USA
{cheng, berzins, luqi, swapan}@cs.nps.navy.mil

Abstract. This paper concentrates on the issues related to implementation of interoperability between distributed subsystems, particularly in the context of re-engineering and integration of several centralized legacy systems. Currently, most interoperability techniques require the data or services to be tightly coupled to a particular server. Furthermore, as most programmers are trained in designing stand-alone application, developing distributed system proves to be time-consuming and difficult. Here, we addressed those concerns by creating an interface wrapper model that allows developers to treat distributed objects as local objects. A tool that automatically generates the features of Java interface wrapper from a specification language called the Prototyping System Description Language has been developed based on the model.

1 Introduction

Interoperability between software systems is the ability to exchange services from one system to another. In order to exchange services, commands and data are relayed from the requesters to the service providers. Current business and military systems are typically 2-tier or 3-tier systems involving clients and servers, each running on different machines in the same or different locations. Current approaches for n-tier systems have no standardization of protocol, data representation, invocation techniques etc. Other problems related to interoperability are the implementation of distributed systems and the use of services from heterogeneous operating environments. These include issues concerning sharing of information amongst various operating systems, and the necessity for evolution of standards for using data of various types, sizes and byte ordering, in order to make them suitable for interoperation. These problems make interoperable applications difficult to construct and manage.

1.1 Current State-of-the-Art Solutions

Presently, the solutions attempting to address these interoperability problems range from low-level sockets and messaging techniques to more sophisticated middleware technology like object resource brokers (CORBA, DCOM). Middleware technology uses higher abstraction than messaging, and can simplify the construction of interoperable applications. It provides a bridge between the service provider and

requester by providing standardized mechanisms that handle communication, data exchange and type marshalling. The implementation details of the middleware are generally not important to developers building the systems. Instead, developers are concerned with service interface details. This form of information hiding enhances system maintainability by encapsulating the communication mechanisms and providing stable interface services for the developers. However, developers still need to perform significant work to incorporate the middleware's services into their systems. Furthermore, they must have a good knowledge of how to deploy the middleware services to fully exploit the features provided.

Current middleware approaches have another major limitation in design – the data and services are tightly coupled to the servers. Any attempt to parallelize or distribute a computation across several machines therefore encounters complicated issues due to this tight control of the server process on the data. Tuning performance by redistributing processes and data over different hardware configurations requires much more effort for software adjustment than system administrators would like.

1.2 Motivation

Distributed data structures provide an entirely different paradigm. Here, data is no longer coupled to any particular process. Methods and services that work on the data are also uncoupled from any particular process. Processes can now work on different pieces of data at the same time. Until recently, building distributed data structures together with their requisite interfaces has proved to be more daunting than other conventional interoperability middleware techniques. The arrival of JavaSpace has changed the scenario to some extent. It allows easy creation and access to distributed objects. However, issues concerning data getting lost in the network, duplicated data items, out-dated data, external exception handling and handshaking communication between the data owner and data users are still open. Developers have to devise ways to solve those problems and standardize them between applications.

1.3 Proposal

The situation concerning interoperability would greatly improve if a developer working on some particular application could treat distributed objects as local objects within the application. The developers could then modify the distributed object as if it is local within the process. The changes may, however, still need to be reflected in other applications using that distributed object without creating any problems related to inconsistency. The current research aims at attaining this objective by creating a model of an interface wrapper that can be used for a variety of distributed objects. In addition, we seek models that can automate the process of generating the interface wrapper directly from the interface specification of the requirement, thereby greatly improving developers' productivity.

A tool, named the Automated Interface Codes Generator (AICG), has been developed to generate the interface wrapper codes for interoperability, from a specification language called the Prototype System Description Language (PSDL) [9]. The tool

uses the principles of distributed data structure and JavaSpace Technology to encapsulate transaction control, synchronization, and notification together with lifetime control to provide an environment that treats distributed objects as if there were local within the concerned applications.

2   Review of Previous Works

A basic idea for enhancing interoperability is to make the network transparent to the application developers. Previous approaches [1] include 1) Building blocks for interoperability, 2) Architectures for unified, systematic interoperability and 3) Packaging for encapsulating interoperability services. These approaches have been assessed and summerized using Kiviat graphs by Berzins [1] with various weight factors. The Kiviat graphs give a good summary of the strong and weak points of various approaches. ORBs and Jini are currently among the promising technologies for interoperability.

2.1   ORB Approaches

There are however, some concerns with the ORB models. Sullivan [13] provides a more in-depth analysis of the DCOM model, highlighting the architecture conflicts between Dynamic Interface Negotiation (how a process queries a COM interface and its services) and Aggregation (component composition mechanism). Interface negotiation does not function properly within the aggregated boundaries. This problem arises because interacting components share an interface. An interface is shared if the constructor or QueryInterface functions of several components can return a pointer to it. QueryInterface rules state that a holder of a shared interface should be able to obtain interfaces of all types appearing on both the inner and outer components. However, an aggregator can refuse to provide interfaces of some types appearing on an inner component by hiding the inner component. Thus, QueryInterface can fail to work properly with respect to delegation to the inner interface.

Hence, for the ORB approaches, detailed understanding of the techniques is required to design a truly reliable interoperable system. Programmers however, are trained mostly on standalone programming techniques. Adding specialized network programming models increases the learning as well as development time, with occasional slippage of target deadlines. Furthermore, bugs in distributed programs are harder to detect and consequences of failure are more catastrophic. An abnormal program can cause other programs to go astray in a connected distributed environment [9], [12].

2.2   Prototyping

The demand for large, high quality systems has increased to the point where a quantum change in software technology is needed [9]. Requirements and

specification errors are a major cause of faults in complex systems. Rapid prototyping is one of the most promising solutions to this problem. Completely automated generation of prototype from a very high-level language is feasible and generation of skeleton programming structures is currently common in the computer world. One major advantage of the automatic generation of codes is that it frees the developers from the implementation details by executing specifications via reusable components [9].

In this perspective, an integrated software development environment, named Computer Aided Prototyping System (CAPS) has been developed at the Naval Postgraduate School, for rapid prototyping of hard real-time embedded software systems, such as missile guidance systems, space shuttle avionics systems, software controllers for a variety of consumer appliances and military Command, Control, Communication and Intelligence (C3I) systems [11]. Rapid prototyping uses rapidly constructed prototypes to help both the developers and their customers visualize the proposed system and assess its properties in an iterative process. The heart of CAPS is the Prototyping System Description Language (PSDL). It serves as an executable prototyping language at a specification and software architecture level and has special features for real-time system design. Building on the success of computer aided rapid prototyping system (CAPS) [11], the AICG model also uses PSDL for specification of distributed systems and automates the generation of interface codes with the objective of making the network transparent from the developer's point of view.

2.3 Transaction Handling

Building a networked application is entirely different from building a stand-alone system in the sense that many additional issues need to be addressed for smooth functioning of a networked application. The networked systems are also susceptible to partial failures of computation, which can leave the system in an inconsistent state.

Proper transaction handling is essential to control and maintain concurrency and consistency within the system. Yang has examined the limitation of hard-wiring concurrency control into either the client or the server. He found that the scalability and flexibility of these configurations is greatly limited. Hence, he presented a middleware approach: an external transaction server, which carries out the concurrency control policies in the process of obtaining the data. Advantages of this approach are 1) transaction server can be easily tailored to apply the desired concurrency control policies of specific client applications. 2) The approach does not require any changes to the servers or clients in order to support the standard transaction model. 3) Coordination among the clients that share data but have different concurrency control policies is possible if all of the clients use the same transaction server.

The AICG model uses the same approach, by using an external transaction manager such as the one provided by SUN in the JINI model. All transactions used by the clients and servers are created and overseen by the manager.

3 The AICG Interaction Model

The AICG model encapsulates some of the features of JavaSpace and Jini to provide a simplified ways of developing distributed applications.

3.1 Jini Model

The Jini model is designed to make a service on a network available to anyone who can reach it, and to do so in a type-safe and robust way [4]. The ability of Jini model is based on five key concepts: (1) *Discovery* is the process used to find communities on the network and join with them. (2) *Lookup* governs how the code that is needed to use a particular services finds its way into participants that want to use that service. (3)*Leasing* is the technique that provides the Jini self recovering ability. (4) *Remote events* allow services to notify each other of changes to their state (5) *Transactions* ensure that computations of several services and their host always remain in "safe" state.

The Jini model was designed by Sun Microsystems with simplicity, reliability and scalability as the focus. Its vision is that Jini-enable devices such as PDA, cell phone or a printer, when plugged into a TCP/IP network, should be able to automatically detect and collaborate with other Jini-enabled devices.

The powerful features of Jini provide a good groundwork for developing interoperability systems. However, the lack of automation for creating interface software and the need for developers to fully understand the Jini Model before they can use it created the same problems for developers as other interoperability approaches.

3.2 The JavaSpace Model

The JavaSpace model is a high-level coordination tool for gluing processes together in a distributed environment. It departs from conventional distribution techniques using message passing between processes or invoking methods on remote objects. The technology provides a fundamentally different programming model that views an application as a collection of processes cooperating via the flow of freshly copied objects into and out of one or more spaces. This space-based model of distributed computing or distributed structure has its roots in the Linda coordination language [3] developed by Dr. David Gelernter at Yale University.

3.2.1 Distributed Data Structure and Loosely Coupled Programming
Conceptually a distributed data structure is one that can be accessed and manipulated by multiple processes at the same time without regard for which machine is executing those processes. In most distributed computing models, distributed data structures are hard to achieve. Message passing and remote method invocation systems provide a good example of the difficulty. Most of the systems tend to keep data structure behind one central manager process, and processes that want to perform work on the data

structure must "wait in line" to ask the manager process to access or alter a piece of data on their behalf. Attempts to parallelize or distribute a computation across more than one machine face bottlenecks since data are tightly coupled by the one manager process. True concurrent access is rarely achievable.

Distributed data structures provide an entirely different approach where we uncouple the data from any particular process. Instead of hiding data structure behind a manager process, we represent data structures as collections of objects that can be independently and concurrently accessed and altered by remote processes. Distributed data structures allow processes to work on the data without having to wait in line if there are no serialization issues.

3.2.2 Space

A space is a shared, network-accessible repository for objects. Processes use the repository as a persistent object storage and exchange mechanism Processes perform simple operations to write new objects into space, take objects from space, or read (make a copy of) objects in a space. When taking or reading objects, processes use a simple value-matching lookup to find the objects that matter to them. If a matching object is not found immediately, then a process can wait until one arrives. Unlike conventional object stores, processes do not modify objects in the space or invoke their methods directly. To modify an object, a process must explicitly remove it, update it, and reinsert it into the space. During the period of updating, other processes requesting for the object will wait until the process writes the object back to the space. This protocol for modification ensures synchronization, as there can be no way for more than one process to modify an object at the same time. However, it is possible for many processes to read the same object at the same time.

Key Features of JavaSpace:

- Spaces are persistent: Spaces provide reliable storage for objects. Once stored in the space, an object will remain there until a process explicitly removes it.
- Spaces are transactionally secure: The Space technology provides a transaction model that ensures that an operation on a space is atomic. Transactions are supported for single operations on a single space, as well as multiple operations over one or more spaces.
- Spaces allow exchange of executable content: While in the space, objects are just passive data, however, when we read or take an object from a space, a local copy of the object is created. Like any other local object; we can modify its public fields as well as invoke its methods.

3.3 The AICG Approach

The AICG approach to interoperability has two parts. The first part is to develop a model to completely hide the interoperability from the developers and the second part of the approach is to design a tool that automates the process of integrating the AICG model into the distributed application so as to aid the development process.

3.3.1 The AICG Model

The AICG model is built on JavaSpace and Jini. It is designed to wrap around data structures or objects that are shared between concurrent applications across a network. The model gives the applications complete access to the contents of the objects as though they were the sole owners of the data. Synchronization, transaction and error handling are built into the model, freeing the developers to concentrate on the actual requirement of the applications.

AICG uses the JavaSpace Distributed Data Structure principles as the main communication channel for exchange of services. The model also encompasses Jini services like Transaction, Leasing and Remote Event. However, the difference is that the model wraps the services provided by the JavaSpace and Jini and hide their usage from the application. Developers are not required to understand the underlying principles before they can use the model. They should however be aware of object oriented programming constraints such as no direct access to the attributes of an object is allowed without going through the object methods.

The most common use of the AICG model is to encapsulate objects that are to be shared. This form of abstraction has an advantage over direct use of a JavaSpace. The JavaSpace distributed protocol for modification ensures synchronization by enforcing that a process wishing to modify the object has to physically remove it from the space, alter it and write it back to the space. There can be no way for more than one process to modify an object at the same time. However, this does not prevent other processes from overwriting the updated data. For example, in an ordinary JavaSpace, the programmer of Process A could specify a "read" operation, followed by a "write" operation. This would result in 2 copies of the object in the Space. The AICG model prevents this since the 3 basic commands are embedded into distributed objects that are automatically generated to conform to the proper protocol. All modifications on the object are automatically translated to "take", followed by "write" and all operations that access the fields of the distributed object are translated to "read". These ensure that local data are up-to-date and serialization is maintained.

Although the basic idea of the AICG model is simple, it requires many supporting features to make it work. Distributed objects may be lost if a process removes them from the space and subsequently crashes or is cut off from the network. Similarly, the system may enter a deadlock state if processes request more than one distributed object while, at the same time, holding on to distributed objects required by other processes. Similarly, latency and performance are very different between local access and remote access. Those issues should not be ignored in any interoperability techniques, if the systems to be built using the techniques must be robust. ORB techniques such as RPC and CORBA do not even consider performance and latency as part of their programming model, they treat it as a "hidden" implementation detail that programmer must implicitly be aware of and deal with while they preach that accessing remote object is similar to accessing local object.

The AICG model has a set of four supporting modules to handle those situations. These modules provide transaction handling and user-defined latency to ensure integrity of the updates, exception handling for reporting errors and failures without

crashing the system, a notification channel to inform the application of certain events, and lease control for freeing up unused object during "house keeping". The supporting features are discussed in section 5.

3.3.2 The AICG Tool

The second part of the research aims at developing a tool that generates software wrapper realizing the AICG model to aid the construction of distributed applications. The tool is designed to generate interface wrappers for data structures or objects that need to be shared, and is particularly useful for applications that can be modeled as flows of objects through one or more servers. The tool allows the developers to use all the features in the AICG model without the need to write complicated codes. This enhances interoperability by making network and concurrent issues transparent to the application developers.

The interface wrappers are generated from an extension of a prototype description language called Prototyping System Description Language (PSDL). The extended Description language (PSDL-ext) expands property definitions that are specific only to AICG model.

Some of the salient features of the AICG model generated by the tool are:

- Distributed objects are treated as local objects within the application process. The application code need not depend on how the object is distributed, since the local object copy is always synchronous with the distributed copy.
- Synchronization with various applications is automatically handled. Since the AICG model is based on the space transaction secure model and all operations are atomic. Deadlock is prevented automatically within the interface and each object has through transaction control. Any type of object can be shared as long as the object is serializable. Any data structure and object can be distributed as long as it obeys and implements the java serializable feature.
- Every distributed object has a lifetime. The distributed object lifetime is a period of time guaranteed by the AICG model for storage and distribution of the object. The time can be set by developer.
- All write operations are transaction secure by default. AICG transactions are based on the Atomicity, Consistency, Isolation, and Durability (ACID) features.
- Clients can be informed of changes to the distributed object through the AICG event model. A client application can subscribe for change notification, and when the distributed object is modified, a separate thread is spawned to execute a callback method defined by the developer.
- The wrapper codes are generated from high-level descriptive languages; hence, they are more manageable and more maintainable.

4 Types of Services

Services can be basic raw data, messages, remote method invocation, complex data structures, or object with attributes and methods. The AICG model is suited for exchange and sharing of complex data structures and objects. It can be tailored for raw data, messaging, and remote method invocation types of communication.

The AICG model uses the space as a transmission medium and hence loosens the tie between producers and consumers of services which are forced to interact indirectly through a space. This is a significant difference, as loosely coupled systems tend to be more flexible and robust.

4.1 Overview of the PSDL Interface

Prototype System Description Language (PSDL) provides a data flow notation augmented by application-orientated timing and control constraints to describe a system as a hierarchy of networks of processing units communicating via data streams [1]. Data Streams carry values of abstract types and provide error-free communication channels. PSDL can be presented in a semi-graphical form for easy specifying of the specifications and requirements. An introduction to the real-time aspects of the PSDL can be found in [1] and [2].

In PSDL, every computational entity such as an application, a procedure, a method or a distributed system is represented as an operator. It is hierarchical in nature and each operator can be decomposed to sub-operators and streams. Every operator is a state machine. Its internal states are modeled by variable sets local only to this operator. Operators are represented as circular icons in PSDL Graph, and triggered by data stream or periodic timing constraints. When an operator is triggered, it reads one data value from each input stream and computes the results if the execution guard or constraint is satisfied. The results are placed on the output streams if the output guard is satisfied.

Operators communicate via data streams. These data streams contain values that are instances of an abstract data type. For each stream, there are zero or more operators that write data on the stream and zero or more operators that read data from that stream. There are two kinds of streams in PSDL, *dataflow* and *sampled streams*. *Dataflow* streams act as FIFO buffers, where the data values cannot be lost or replicated. These streams are used to synchronize data from multiple sources. Consumers of dataflow streams never read an empty stream. Similarly, each value in a stream is read only once. The control constraint used by the PSDL to distinguish a stream as dataflow is "TRIGGERED BY ALL".

Sampled Streams act as atomic memory cells providing continuous data. Connected operators can write on or read from the streams at uncoordinated rates. Older data are lost if the producer is faster than the consumer. Absence of "TRIGGERED BY ALL" control constraint implies the stream is sampled.

If any of the streams have any initial value, then it is known as *State Stream*. State Streams are declared in specification of the parent operator and are represented by thicker lines in the PSDL graph. State streams correspond to spaces that contain objects intended to be updated.

The mapping of dataflow streams or sampled streams into space-based communication is accomplished by treating the services, which in this case are the communication streams as objects to be shared.

4.2 Benefit of Loosely Coupled Communication

In tightly coupled systems, the communication process needs the answers to the questions of "who" to send to, "where" the receiving parties are located, and "when" the messages need to be sent. The "who" is which processes, "where" is which machines, and "when" is right now or later. They must be specified explicitly in order for the message to be delivered. Hence, in a distributed environment, in order for a producer and consumer to communicate successfully, they must know each other's identity and location, and must be running at the same time. This tight coupling leads to inflexible applications that are not mobile and in particular difficult to build, debug and change. In loosely coupled systems the issues of "who?", "where?" and "when?" are answered with "anyone", "anywhere" and "anytime".

"Anyone": Producers and consumers do not need to know each other's identities, but can instead communicate anonymously. In the sampled stream mapping, the producers place a message entity into the space without knowing who will be reading the messages. Similarly, the consumers read the message entity from the space without concern with the identity of the producers.

"Anywhere": Producers and consumers can be located anywhere, as long as they have access to an agreed-upon space for exchanging messages. The producer does not need to know the consumer's location. Conversely, the consumer picks up the message from the space using associative lookup, and has no need to be aware of the producer location. This is especially useful when the producers and the receivers roam from machine to machine, because the space-based programs do not need to change.

"Anytime": With space-based communication, producers and consumers are able to communicate even if they do not exist at the same time, because message entries persist in the space. This works well when the producers and the consumers operate asynchronously (Sampled Stream). This does not mean that synchronous communication would not work; the space is also an event driven repository and can trigger the consumers whenever new entities are created in the space. This decoupling in time is useful because it enables operators to be scheduled flexibly to accommodate real-time constraints.

5 How AICG Unifies Localized and Distributed Systems

The AICG model aims at bridging the differences between localized and distributed systems by simplifying the distributed model and encapsulating all the necessary elements of the distributed systems into the wrapper interfaces.

5.1 Localized and Distributed Systems

The major differences between localized and distributed systems concern the areas of latency, memory access, partial failure, and concurrency. Most of interoperability techniques try to hide the network and simplify the problems by stating that locations

of the software components do not affect the correctness of the computations, just the performance. These techniques concentrate on addressing the packing of data into portable forms, causing an invocation of a remote method somewhere on the network and so forth. However, latency, performance, partial failure and concurrency are some of the characteristics of distributed systems which also need serious attention.

5.1.1 Latency and Memory Access

The most obvious difference between accessing a local object and accessing a remote object has to do with the latency of the two calls. The difference between the two is currently between four and five orders of magnitude. In the AICG model vision of unified object where remote access is actually a three steps process, step one retrieves remote object from the space, step two executes the method of the remote object locally and lastly step three returns the object back to the space if it is modified. Developers must be aware of the latency and performance concerns. To ensure that the developers are aware of the issues, the AICG model requires the developers to specify the maximum latency period before an exception is raised. This forces the developers to consider the latency issues for the type of data and methods that are to be shared.

Another fundamental difference between local and remote computing concerns access to memory, specifically in the use of pointers. Simply stated, pointers are valid only within the local address space. There are two solutions; either all the memory access must be controlled by the underlying system, or the developers must be aware of the different type of access, whether local or remote.

Using the object-oriented paradigm to the fullest is a way of eliminating the boundary between the local and remote computing. However, it requires the developers to build an application that is entirely object-oriented. Such a unified model is difficult to enforce. The AICG solution to this issue is by enforcing the object-oriented paradigm only on distributed objects. The distributed object wrapper generated automatically forces all access to the actual shared object to go through the wrapper which is always a local object, eliminating direct reference to the actual object itself. This promotes and enforces the principle that "remote access and local access are exactly the same".

5.1.2 Partial Failure and Concurrency

In case of local systems, failures are usually total, affecting all the components of the system working together in an application. In distributed systems; one subsystem can fail while other systems continue. Similarly, a failure of network link is indistinguishable from the failure of a system on the other end of the link. The system may still function with partial failure, if certain unimportant components have crashed. It is however difficult to detect partial failure since there is no common agent that is able to determine which systems have failed, and this may result in the entire system going into unstable states

The AICG model uses the loosely-coupled paradigm, and component failure may have impact on the distributed system when the systems retrieve objects from the space and later crash before returning the objects back to space. The AICG model resolves this issue by enforcing update of distributed objects with transaction control

and allowing the developers to specify useful lifetime or lease for the object. When a lease has expired, the object would be automatically removed from the space.

Distributed objects by their nature must handle concurrent access and invocations. Invocations are usually asynchronous and difficult to model in distributed systems. Usually most models leave the concurrency issues to the developers discretion during implementation. However, this should be an interface issue and not solely an implementation issue, since dealing with concurrency can take place only by passing information from one object to another through the agency of the interface. The AICG model handles concurrency by design since there is only one copy of distributed object at a time in the entire distributed system. Processes are made to wait if the shared objects are not available in the space.

5.2   Transaction

Transaction control must validate operations to ensure consistency of the data, particularly when there are consistency constraints that link the states of several objects. The AICG model implements the transaction feature with the Jini Transaction model and provide a simplified interface for the developers.

5.2.1   Jini Transaction Model

All transactions are overseen by a transaction manager. When a distributed application needs operations to occur in a transaction secure manner, the process asks the transaction manager to create a transaction. Once a transaction has been created, one or more processes can perform operations under the transaction. A transaction can complete in two ways. If a transaction commits successfully, then all operations performed under it are complete. However, if problems arise, then the transaction is aborted and none of the operations occur. These semantics are provided by a two-phase commit protocol that is performed by the transaction manager as it interacts with the transaction participants.

5.2.2   AICG Transaction Model

AICG model encapsulates and manages the transaction procedures. All operations on a distributed object can be either with transaction control or without. Transaction control operations are controlled with a default lease of six sec. This default value of leasing time may, however, be overridden by the user. This is kept by the transaction manager as a leased resource, and if a lease expires before the operation committed, the transaction manager aborts the transaction.

The AICG model by default, enables all transactions for *write* operations with a transaction lease time of six seconds. The developer can modify the lease time through the PSDL SPACE *transactiontime* property.

All the read operations in the AICG model do not have transactions enabled by default. However, the user can enable it by using the property transactiontime with the upper limit in transaction time for the read operation.

5.3   Object Life Time (Leases/Timeout)

Leasing provides a methodology for controlling the life span of the distributed objects in the AICG space. This allows resources to be freed after a fixed period. This model is beneficial in the distributed environment, where partial failure can cause holders of resources to fail thereby disconnecting them from the resources before they can explicitly free them. In the absence of a leasing model, resource usage could grow without bound.

There are other constructive ways to harness the benefit of the leasing model besides using it as a garbage collector. For example, in a real-time system, the value of the information regarding some distributed objects becomes useless after certain deadlines. Accessing obsolete information can be more damaging in this case. By setting the lease on the distributed object, the AICG model automatically removes the object once the lease expires or the deadline is reached.

Java Spaces allocate resources that are tied to leases. When a distributed object is written into a space, it is granted a lease that specifies a period for which the space guarantees its storage. The holder of the lease may renew or cancel the lease before it expires. If the leaseholder does neither, the lease simply expires, and the space removes the entry from its store.

Generally, a distributed object that is not a part of a transaction lasts forever as long as the space exists, even if the leaseholder (the process that creates the object) has died. This configuration is enabled by setting the *SPACE lease* property in the Implementation to 0.
In real-time environment, a distributed object lasts for a fixed duration of x ms specified by the object designer. To keep the object alive, a write operation must be performed on the object before the lease expires. This configuration is set through the *SPACE lease* property in the Implementation to the time in ms required.

If an object has a lifetime, it must be renewed before it expires. In the AICG model, renewal is achieved by calling any method that modifies the object. If no modification is required, the developer can provide a dummy method with the spacemode set to "write". Invoking that method will automatically renew the lease.

5.4   AICG Event Notification

In a loosely-coupled distributed environment, it is desirable for an application to react to changes or arrival of newly distributed objects instead of "busy waiting" for it through polling. AICG provides this feature by introducing a callback mechanism that invokes user-defined methods when certain conditions are met.

Java provides a simple but powerful event model based on event sources, event listeners and event objects. An event source is any object that "fires" an event, usually based on some internal state change in the object. In this case, writing an object into space would generate an event. An event listener is an object that listens for events fired by an event source. Typically, an event source provides a method whereby

listeners can request to be added to a list of listeners. Whenever an event source fires an event, it notifies each of its registered listeners by calling a method on the listener object and passing it an event object.

Within a Java Virtual machine (JVM), an application is guaranteed that it will not miss an event fired from within. Distributed events on the other hand, had to travel either from one JVM to another JVM within a machine or between machines networked together. Events traveling from one JVM to another may be lost in transit, or may never reach their event listener. Likewise, an event may reach its listener more than once.

Space-based distributed events are built on top of the Jini Distributed Event model, and the AICG event model further extends it. When using the AICG event model, the space is an event source that fires events when entries are written into the space matching a certain template an application is interested in. When the event fires, the space sends a remote event object to the listener. The event listener codes are found in one of the generated AICG interface wrapper files. Upon receiving an event, the listener would spawn a new thread to process the event and invoke the application callback method. This allows the application codes to be executed without involving the developer in the process of event-management.

The distributed objects must have the SPACE properties for *Notification* set to yes. One of the application classes must *implement* (java term for inherit) the notifyAICG abstract class. The notifyAICG class has only one method, which is the callback method. The user class must override this method with the codes that need to be executed when an event fires.

6 Developing Distributed Application with the AICG Tool

This section describes the steps for developing distributed applications using the AICG model. An example of a C4ISR application is introduced in section 6.2 to aid the explanation of the process.

6.1 Development Process

The developer starts the development process by defining shared objects using the Prototyping System Description Language (PSDL). The PSDL is processed through a code generator (PSDLtoSpace) to produce a set of interface wrapper codes. The interface wrappers contain the necessary codes for interaction between application and the space without the need for the developers to be concerned with the writing and removing of objects in the space. The developers can treat shared or distributed objects as local objects, where synchronization and distribution are automatically handled by the interface codes.

6.2   Input Definition to the Code Generator

The following example demonstrates the development of one of the many distributed objects in a C4ISR system. Airplane positions picked up from sensors are processed to produce track objects. These objects are distributed over a large network and used by several clients' stations for displaying the positions of planes. Each track or plane is identified by track number. The tracks are 'owned' by a group of track servers, and only the track servers can update the track positions and its attributes. The clients only have read access on the track data. PSDL codes define (1) track object and as well as (2) Track_list object with the corresponding methods. AICG has used an extended version of the original PSDL grammar to model the interactions between applications in an entire distributed system.

The track PSDL starts with the definition of a *type* called **track**. It has only one identification field **tracknumber**. Of course, the **track** objects can have more than one field, but only one field is used in this case to uniquely identify any particular **track** object. The type **track_list** on the other hand, does not need an identification field since there is only one **track_list** object in the whole system. **Track_list** is used to keep a list of the **tracknumbers** of all the active tracks in the system at each moment in time.

All the operators (methods) of the *type* are defined immediately after the specification. Each method has a list of *input* and *output* parameters that define the arguments of the method. The most important portion in the method declaration is the *implementation*. The developer must be able to define the type of operation the method supposed to perform. The operation types are *constructor* (used to initialize the class), *read* (no modification to any field in the class) and *write* (modification is done to one or more fields in the class). These are necessary, as the code generated will encapsulate the synchronization of the distributed objects.

The other field in the implementation portion of the method, is *transactiontime*. *transactiontime* defines the upper limit in milliseconds within which the operation must be completed.

Upon running the example through the generator tool, a set of Java interface wrapper files are produced. Developers can ignore most of the generated files except the following:

- Track.java: this file contains the skeleton of the fields and the methods of the track class. The user is supposed to fill the body of the methods.
- TrackExtClient.java: this is the wrapper class that the client initializes and uses instead of the track class.
- TrackExtServer.java: this is the wrapper class that the server initializes and uses instead of the track class.
- NotifyAICG.java : this class must be extended or implemented by the application if event-notification and call-back are needed.

The methods found in the trackExtClient and trackExtServer have the same method names and signatures as the track class. In fact, the track class methods are called within trackExtClient or trackExtServer.

7   AICG Wrapper Design

This section explains the design of the AICG and the codes that are generated from psdl2java program.

7.1   AICG Wrapper Architecture

The AICG wrapper codes generated consists of four main module types. They are the Interface modules, the Event modules, Transaction modules and the Exception module. The interface modules implement the distributed object methods and communicate directly with the application. In reference to the example in section 6.2, the interface modules are entryAICG, track, trackExt, trackExtClient, trackExtServer. Instead of creating the actual object (track), the application should instantiate the corresponding interface object, either the trackExtClient or trackExtServer. Event modules (eventAICGID, eventAICGHandler, notifyAICG) handle external events generated from the JavaSpace that are of interest to the application. Transaction modules (transactionAICG, transactionManagerAICG) support the interface module with transaction services. Lastly, the exception module (exceptionAICG) defines the possible types of exceptions that can be raised and need to be captured by the application.

Each time the application instantiates a track class by creating a new trackExtServer, the following events take place in the Interface:
1. An Entry object is created together with the track object by the trackExtServer. The tack object is placed into the Entry object and stored in the space.
2. Transaction Manager is enabled.
3. The reference pointer to trackExtServer is returned to the application.

Each time a method (getID, getCallsign, getPosition) that does not modify the contents of the object is invoked, the following events take place in the Interface:
1. The application invokes the method through the Interface (trackExtServer/trackExtClient).
2. The Interface performs a Space "get" operation to update the local copy.
3. The method is then executed on the updated copy of the object to return the value back to the application.

Each time a method (setCallsign, setPosition), which does modify the contents of the object is invoked, the following events take place in the Interface:
1. The application invokes the method through the Interface.
2. The interface performs a Space "take" operation, which retrieves the object from the space.
3. The actual object method is then invoked to perform the modification.

41

4. Upon completion of the modification, the object is returned to the space by the interface using a "write" operation.

7.2 Interface Modules

The interface modules consist of the following modules; an entry (entryAICG) that is stored in space, the actual object (trackExt) that is shared and the object wrapper (trackExt, trackExtClient, trackExtServe.).

7.2.1 Entry
A space stores *entries*. An entry is a collection of typed objects that implements the Entry interface. The Entry interface is empty; it has no methods that have to be implemented. Empty interfaces are often referred to as "marker" interfaces because they are used to mark a class as suitable for some role. That is exactly what the Entry interface is used for, to mark a class appropriate for use within a space.

All entries in the AICG extend from this base class. It has one main public attribute, an identifier and an abstract method that returns the object. Any type of object can be stored in the entry. The only limitation is that the object must be serializable. The serializable property allows the java virtual machine to pass the entire object by value instead of by reference

All Entry attributes are declared as publicly accessible. Although it is not typical of fields to be defined as public in object-oriented programming style, an associative lookup is the way the space-based programs locate entries in the space. To locate an object in space, a template is specified that matches the contents of the fields. By declaring entry fields public, it allows the space to compare and locate the object. AICG encourages object-oriented programming style by encapsulating the actual data object into the entry. The object attributes can then be declared as private and made accessible only through clearly defined public methods of the object.

7.2.2 Serialization
Each distributed interface object is a local object that acts as a proxy to the remote space object. It is not a reference to a remote object but instead a connection passes all operations and value through the proxy to the remote space. All the objects must be serializable in order to meet this objective. The Serializable interface is "marker" interface that contains no methods and serves only to mark a class as appropriate for serialization. Classes marked as serializable should not contain pointers in their representation.

7.2.3 The Actual Object
We now look at the actual objects that are shared between servers and clients. The psdl2java generates a skeleton version of the actual class with the method names and its arguments. The bodies of the methods and its fields need to be filled by the developers.

7.2.4 Object Wrapper

Wrapping is an approach to protecting legacy software systems and commercial off-the-shelf (COTS) software products that require no modification of those products [1]. It consists of two parts, an adapter that provides some additional functionality for an application program at key external interfaces, and an encapsulation mechanism that binds the adapter to the application and protects the combined components [1].

In this context, the software being protected contains the actual distributed objects, and the AICG model has no way of knowing the behaviors of the distributed objects other than the operation types of of the methods. The adapter intercepts all invocations to provide additional functionalities such as synchronization between the local and distributed object, transaction control, event monitoring and exception handling. The encapsulation mechanism has been explained in the earlier section (AICG Architecture). Instead of instantiation of the actual object, the respective interface wrapper is instantiated. Instantiating the interface wrapper indirectly instantiates the actual object as well as storing the object in the space.

Three classes are generated for every distributed object. There are named with the object name appended with the following Ext, ExtClient, and ExtServer.

7.3   Event Modules

The event modules consist of the event callback template (notifyAICG), the event handler (eventAICGHandler) and the event identification object (eventAICGID).

7.3.1 Event Identification Object

The event identification object is used to distinguish one event from others. When an event of interest is registered, an event identification object is created to store the identification and event source. The object has only two methods, an 'equals' method that checks if two event identification objects are the same and a 'to string' method which is used by the event handler for retrieving the right event objects from the hash table.

7.3.2 Event Handler

Event Handler is the main body of the event operation in the AICG model. It handles registration of new events, deletion of old events, listening for event and invoking the right callback for that event. Inside the event handler are in fact, three inner classes to perform the above functions. Events are stored in a hash table with the event identification object as the key to the hash table. This allows fast retrieval of the event object and the callback methods.

The event handler listens for new events from the space or other sources. When an object is written to the space, an event is created by the space and captured by the all the listeners. The event handler would immediately spawn a new thread and check whether the event is of interest to the application.

43

7.3.3 The Callback Template

The callback template is a simple interface class with an abstract method listenerAICGEvents. Its main function is to allow the AICG model to invoke the application program when certain events of interest is "fired". As explained earlier, the notifyAICG interface needs to be implemented by each application that wishes to have notification.

7.4 The Transaction Modules

The transaction modules consist of a transaction interface (transactionAICG) and the transaction factory (transactionManagerAICG).

The transaction interface is a group of static methods that are used for obtaining references to the transaction manager server somewhere on the network. It uses the Java RMI registry or the look-up server to locate the transaction server.

The transaction factory uses the transaction interface to obtain the reference to the server, which is then used to create the default transaction or user-defined transactions. In short the transaction factory can perform the following:

1. Invoke the transaction interface to obtain a transaction manager.
2. Create a default transaction with lease time of 6 seconds.
3. Create a transaction with a user defined lease time.

7.5 The Exception Module

The exception module defines all the exception codes that are returned to the application when certain unexpected conditions occur in the AICG model. The exceptions include

- "UnDefinedExceptionCode"; unknown error occur.
- "SystemExceptionCode"; system level exceptions, such disk failure, network failure.
- "ObjectNotFoundException"; the space does not contain the object.
- "TransactionException"; transaction server not found, transaction expired before commit.
- "LeaseExpiredException"; object lease has expired.
- "CommunicationException"; space communication errors.
- "UnusableObjectException"; object corrupted.
- "ObjectExistsException"; there another object with the same key in the space.
- "NotificationException"; events notification errors.

8 Conclusion

The AICG vision of distributed object-oriented computing is an environment in which, from the developer's point of view, there is no distinct difference between

sharing of objects within an address space and objects that are on different machines. The model takes care of underlying interoperability issues by taking into account network latency, partial failure and concurrency. Automating the generation of interface wrappers directly from the Prototype System Description Language further enhances the reliability of the systems by enforcing proper object-oriented programming styles on the shared objects. Usage of PSDL for specification of shared objects also results in increased efficiency and shorter development time.

References

1. Valdis Berzins, Luqi, Bruce Shultes, Jiang Guo, Jim Allen, Ngom Cheng, Karen Gee, Tom Nguyen, and Eric Stierna : Interoperability Technology Assessment for Joint C4ISR Systems. Naval Postgraduate School Report NPSCS-00-001 September, (1999).
2. Nicholas Carriero, David Gelernter : How to Write Parallel Programs: A Guide to the Perplexed. ACM Computing Surveys, September (1989) 102-122.
3. David Gelernter : Generative Communication in Linda. ACM Transaction on Programming Languages and Systems, Vol. 7, No. 1, January (1985) 80-112.
4. Bill Joy : The Jini Specification. Addison Wesley, Inc. (1999)
5. Edward Keith : Core Jini. Prentice Hall, PTR, (1999)
6. Eun-Gyung Kim : A Study on Developing a Distributed Problem Solving System. IEEE Software, January (1995) 122-127
7. Fred Kuhns, Carlos O'Ryan, Douglas Schmidt, Ossama Othman, Jeff Parsons : The Design and Performance of a Pluggable Protocols Framework for Object Request Broker Middleware. IFIP 6th International Workshop on Protocols For High-Speed Networks (PfHSN' 99), August 25-27, (1999)
8. David Levein, Sergio Flores-Gaitan, Douglas Schmidt : An Empirical Evaulation of OD Endsystem Support for Real-time CORBA Object Request Brokers. Multimedia Computing and Network 2000, January (2000).
9. Luqi, Valdis Berzins : Rapidly Prototyping Real-Time Systems. IEEE Software, September (1988) 25-35
10. Luqi, Valdis Berzins, Bernd Kraemer, Laura White : A Proposed Design for a Rapid Prototyping Language. Naval Postgraduate School Technical Report, March (1989)
11. Luqi, Mantak Shing : CAPS – A Tool for Real-Time System Devleopment and Acquisition. Naval Research Review, Vol 1 (1992) 12-16
12. Luqi, Valdis Berzins, Raymond Yeh : A Prototyping Language for Real-Time Software. IEEE Software, October (1998) 1409-1423
13. Kevin Sullivan, Mark Marchukov, John Socha : Analysis of a Conflict Between Aggregation and Interface Negotiation in Microsoft's Component Object Model. IEEE Transactions on Software Engineering, Vol. 25, No. 4, July/August (1999) 584-599
14. Antoni Wolski : LINDA: A System for Loosely Integrated DataBases. IEEE Software, January (1989)66-73.

Computer Aided Prototyping in a Distributed Environment

Jun Ge, Valdis Berzins, Luqi
Department of Computer Science
Naval Postgraduate School
Monterey, CA 93943
U.S.A.

Abstract[+]

Previous work on computer-aided prototyping system (CAPS) is stepping into a distributed environment to meet the requirement of integrating legacy systems in heterogeneous network. A three-module architecture design, including Supporting Database, System Tools and Execution Manager, is proposed in this paper for the distributed CAPS system (DCAPS). By using wrapper/glue technique, different prototyping tools in a heterogeneous environment share the input/output data files for prototypes. The architecture is generalized for the communication among legacy systems for data interchange. DCAPS not only provides a useful tool for distributed real-time system prototyping, but also is a demonstration of distributed system in heterogeneous environment.

Key words: software interoperability, fast prototyping, distributed system, multi-agent system

1. Introduction

Computer aided prototyping has been found useful in software development, especially for large real-time systems. Prototyping provides the capability to accurately simulate requirements in new application areas. Previous work such as the Computer Aided Prototyping System (CAPS) has demonstrated real-time issues, software reuse and process scheduling in fast prototyping for a single processor computing environment [1-3]. However, it is still hard to make use of existing systems in a distributed environment, especially for real-time systems under a heterogeneous environment. With the fast development of networks and the Internet, interoperability has become the focus of current research. This paper extends research on CAPS to distributed and network computing.

Distributed real-time software system prototyping and interoperability in a heterogeneous environment form the focus of this paper. In recent years, hard real-time, soft real-time and embedded systems are increasingly important in various application areas from e-business to military applications. These systems have strict requirements on accuracy, safety and reliability. Usually such software is large and built on several legacy systems to make use of the partial or full functionalities of these legacy systems. When the legacy systems are physically located in a distributed network, they are connected through certain network protocols. Fast prototyping of these systems helps the users in analysis, design, implementation, verification, validation and optimization. Approaches for modeling, realizing, reconfiguring and allocating logical processes and interactions to processors and communication links are needed to make prototyping useful in this domain.

This paper describes a distributed CAPS system (DCAPS) to fulfill the requirements for distributed software prototyping. Prototype System Description Language (PSDL), a prototyping language, is applied in the description of the real-time software in DCAPS system. PSDL provides the specifications not only for real-time constraints, but also for the connection and interaction among software components. PSDL has open syntax for the design of new features that arise in the context of distributed computing. Wrapper and glue technology is applied for the normalization and data transfer of legacy systems. A multi-agent technique is used to manage the execution process.

Section 2 introduces the three-module architecture of DCAPS system. All the modules are described in detail in Section 3, 4 and 5 separately. Section 6 gives a simple example prototype in DCAPS.

2. System architecture

Earlier work on computer-aided prototyping system (CAPS) uses PSDL, a prototype description language, to describe the real-time software [4]. PSDL itself has an open structure so that the user is able to define new properties for software components, such as new-added network configurations. CAPS prototypes a software system in the following steps. First, user selects the software components from the reusable component libraries to construct the prototype in a graphic editor. This prototype is saved as a plain text file in PSDL format. User may also use the graphic user interface (GUI)

generator provided by CAPS to create the new GUI interface for the prototype. Then, the translator and scheduler work on this PSDL file to generate the wrapper/glue code and dynamic/static schedules respectively. Both the source code of reusable components and automatic generated source code will be compiled together to get the executable final software. It will be tested in CAPS (simulation) for both the execution correctness and the real-time requirements.

As described above, CAPS consists of various prototyping tools to provide all these functionalities. They play different roles during the prototyping process. For example, the scheduler just needs the information of timing constraints for every component, while the translator does not care about such information other than the network configurations and data type definitions. When new properties are enabled in PSDL description of the prototype, for instance to prototype a networked software, some tools must be updated by new generations while the rest stay the same. Therefore, the architecture of CAPS must consider the evolution of its own components.

CAPS tools were originally developed in SunOS operating system for components which are located on one processor. To consider the user's requirement, the user interface is required to migrate to Windows NT operating system. At the same time, the old operating system is not supported by some new technologies. To avoid the complexity of migrating the whole system to a new operating system, CAPS now has to work in a distributed and heterogeneous environment. A new architecture becomes important for the system. On the other hand, CAPS is required to prototype software systems in distributed and heterogeneous environments. The requirements to develop the distributed CAPS (DCAPS) are consistent for constructing the distributed software prototypes, i.e., DCAPS itself is a demonstration of distributed software construction. A three-module architecture is proposed to design the distributed CAPS system (DCAPS).

From the viewpoint of prototyping procedure, DCAPS can group its tools into three basic modules (Figure 1).
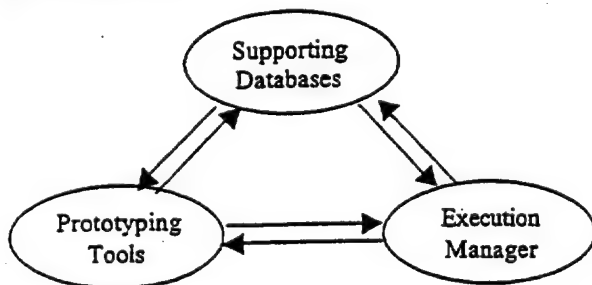

Figure 1. Three-module architecture design of DCAPS

In this architecture, DCAPS provides users support from three aspects. Databases help users to manage and reuse the prototyping requirements and reusable software components. It also validates the prototypes for components' evolution. Prototyping tools help user in automatically generating connection code, GUI code, and data type conversion code among components during the design process. Execution manager controls and visualizes the simulation process to validate the system design, particularly on real-time constraints.

DCAPS inherits prototyping tools that were implemented in different operating systems including SunOS, Solaris and Windows NT. It provides different user interfaces for multiple operating systems including Windows NT. All the tools, which are in the three modules, are located in a distributed environment during one prototyping job.

3. Supporting databases

Supporting databases provide intelligent guidance to users so that in a form of adaptive control it is integrated into the system prototyping. There are two types of database support involved in DCAPS system. One is the software reuse database. It contains the specifications for all the reusable software components so that they are able to be retrieved and to be accessed during the prototyping procedure and the execution (simulation). Software version control should also be considered within this database support. The other is the requirement database. It allows users to reuse the previous prototypes that are stored in the database. Thus it may shorten the design cycle and even optimize the design. The decomposition of this module is shown in Figure 2.
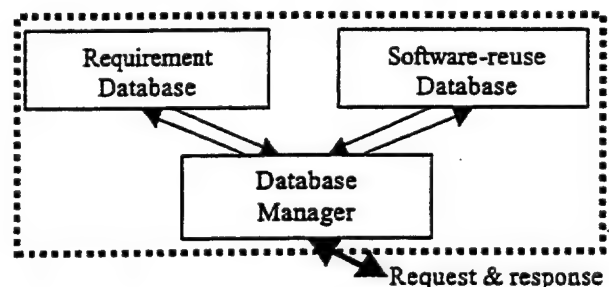

Figure 2. Supporting database system

The browse and retrieve operations for the database includes both syntactic exclusion and semantic exclusion to narrow the search range [5][11].

4. Prototyping tools

Prototyping tools module is decomposed as follows (Figure 3). It includes GUI for various operating systems,

which includes a PSDL graphic editor, the prototype scheduler [9], the prototype translator (automatic code generator for data communication among components), source code compilers and code optimizers for various languages and operating systems. The major operating systems considered in DCAPS are SunOS, Solaris and Windows NT. Job Dispatcher works on a server platform to receive user's commands from GUI and to dispatch jobs to correspondent tools.

The compiler in different operating systems just needs to work with the correspondent automatically generated code. With the change of language in a specific operating system, it is not necessary to change the other components of DCAPS.
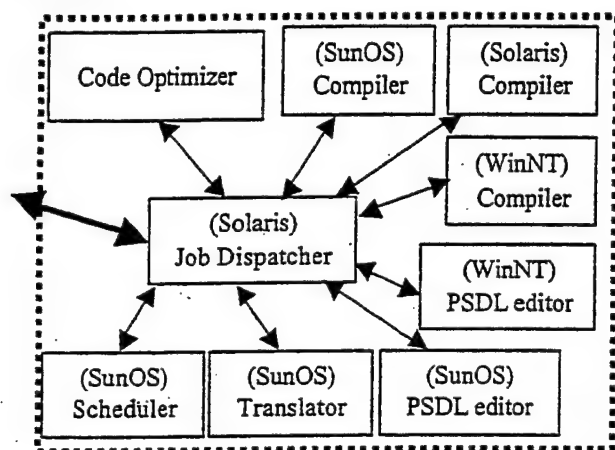

Figure 3. Decomposition of System Tools

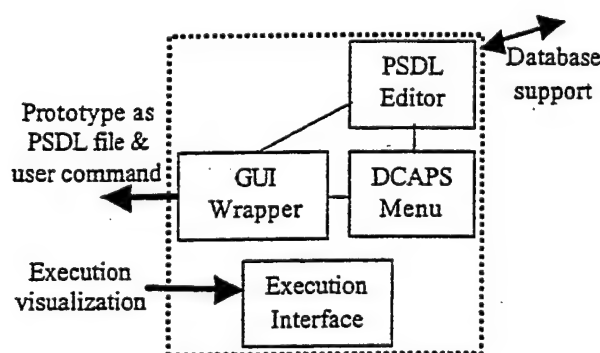The DCAPS GUI can be further decomposed as in Figure 4.

Figure 4. Decomposition of DCAPS GUI

The graphic PSDL editor should be enhanced for new-added properties in the PSDL description of prototype, such as network configuration, different timing constraint, etc. Even in such cases, the system architecture does not have to change at all except that the respective modules are replaced.

The different tools, which are located in different computers, communicate with each other through TCP/IP protocol. The wrapper/glue technique is applied. However, because the data types in communication are known to each other, the wrappers among different tools are blank to each other.

5. Execution manager

The execution of the distributed system, i.e., the simulation of the prototype, is managed by the Execution Manager. It uses a virtual centralized synchronization timer for different task schedules in different processors. This subsystem must compensate for clock drift due to differences in clock rates without violating global timing constraints as long as clock drift rates remain within specified bounds. A multi-agent system is used in the distributed work to coordinate the computing processes.

The Prototyping Scheduler generates one specific task schedule (both dynamic and static) for each node. Execution Manager provides a centralized Executor to administrate and to synchronize the processes in different platforms on which reusable components are located (Figure 5). The procedure of execution is also sent back to GUI of DCAPS so that the user may see a visualized process and have clear information on the prototype.
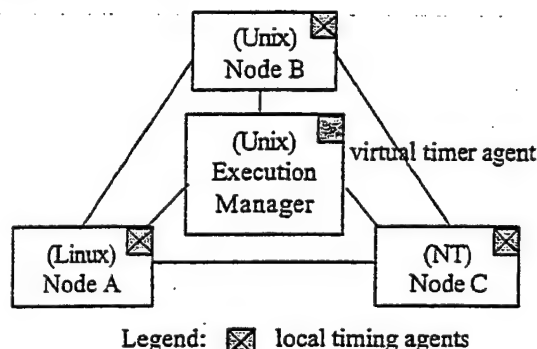
Figure 5. Execution model for a distributed system

In each node, for all the legacy components, the wrapper/glue technology is applied in data interchange (Figure 6). A form of software wrapper and glue technology provides standardized interactions between legacy systems in a heterogeneous network in DCAPS. It makes interoperability and integration possible for a distributed structure. Legacy systems under the wrappers collaborate through the message passing approach in the glue connection. Wrappers provide a generic interface for every single legacy system so that its input and output become uniform, both for consuming data from other legacy systems and for generating data to others. On the other hand, glue structure supports an abstract data class

for data transfer. It encodes any type of data to a common type before putting it into a data stream at the sender's end. At the receiver's end, the data is decoded to the required data type that may be different from that at the sending end. Wrapper and glue concepts are the basis of a formal model for software and hardware co-design.

A multiple-agent system is generated automatically by the Prototyping Translator tool in the architecture as the "glue" for the network communication of the legacy system's inputs and outputs. For each input/output data flow, an agent is associated as an automatic pipe of data transmission. It makes use of the run-time library of network communication according to the specific network protocol in the node that is provided in component information. This "glue" allows the legacy systems not to worry about the network settings for the communication to other components. The communication among agents can reference to several available techniques such as JavaSpace, Jini [7], etc. The technology used in real application should be selected according to the real network configuration.

The "wrapper" code works with the component for data type control/conversion, firing condition, exception handling, timing constraints, etc. The "wrapper" is simply composed in several different layers so that all the features that user concerns are tunable according to user's selections. The "wrapper" communicates to the agents for data outgoing and incoming. Under certain specific conditions, some layer of the wrapper may become transparent based on enhanced information. For example, in the design of DCAPS, the input/output of different prototyping tools are standardized in advance. Therefore, the data type conversion is not required. Because DCAPS itself does not have real-time constraint, the wrapper for timing constraints is transparent.
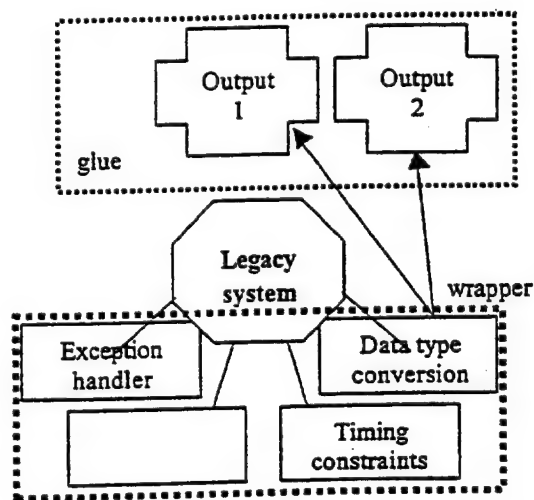
Figure 6. Wrapper/glue architecture for one component

For each processor, a local timing agent manages the execution tasks under the schedule. I/O data of each component is received/sent between legacy system and the uniform software wrapper, which is automatically generated and transferred through glue agents generated by glue code, which hides the specific network configurations via derived design and network mode/parameters.

6. Prototyping example

The system of a weather station is prototyped in DCAPS to demonstrate the ability of prototyping the distributed software in heterogeneous operating system.
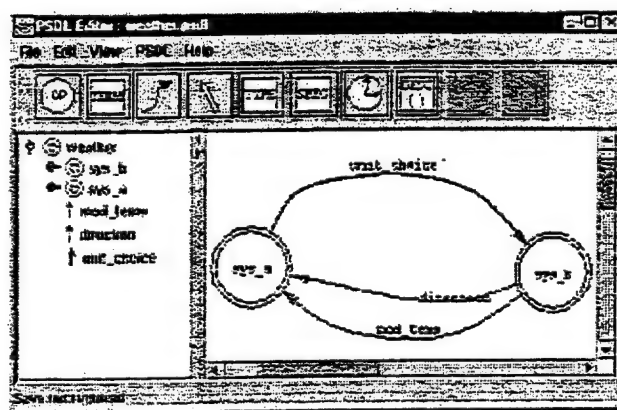
Figure 7. Top level of weather-station prototype
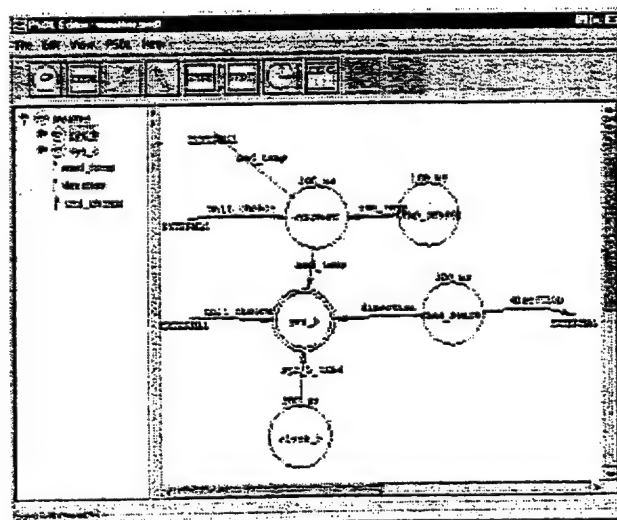
Figure 8. Decomposition of sys_b
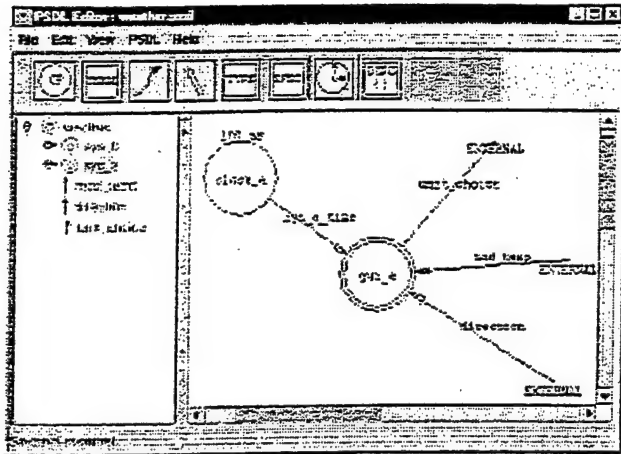
Figure 9. Decomposition of sys_a

Figure 10. Properties configuration for components

As shown in Figure 7-9, weather station system consists of two parts: sys_b is the sensor and sys_a is the controller. The sensor system includes two sub-sensors which are wind direction sensor and temperature sensor. The measurements are converted in specified units. It reports the measurement results to the controller. The controller sends control signal of signal unit to the sensor system so that the sensor can be configured automatically. Both the sub-systems have their own user interfaces in the local systems.

The two sub-systems are located in different computers. They are connected through network in TCP/IP protocol. A SOCKET communication run-time library is provided for data interchange.

DCAPS provides the graphic user interface to edit the prototype in multi-level. For each component, it provides an interface (Figure 10) so that user may specify properties such as timing constraints, network configuration, data flow type, etc. PSDL editor also supports a GUI code generator so that user can create a personal-style user interface for the prototype.

7. Conclusions

The DCAPS system provides a useful tool for distributed real-time software fast prototyping. A three-module architecture is proposed to make DCAPS system suitable for distributed environment. The wrapper/glue method used in DCAPS can be generalized to system construction and interconnection of legacy systems. By automatically generating the codes for the "wrappers and glue" and providing a powerful environment, DCAPS allows the designers to concentrate on the difficult interoperability problems and issues, freeing them from implementation details. It also enables easy reconfiguration of software and network properties to explore design alternatives. DCAPS is an on-going research project for the development and refinement of its prototyping tools.

References

1. Luqi, V. Berzins, Rapidly prototyping real-time systems, IEEE Software, September 1988, pp. 25-36
2. Luqi, W. Royce, Status report: computer-aided prototyping, IEEE Software, Vol. 9, No. 6, November 1992, pp. 77-81
3. Luqi, M. Shing, Real-time scheduling for software prototyping, J. of Systems Integration, special issue on computer-aided prototyping (Vol. 6, No. 1, 1996), pp. 41-72
4. Luqi, V. Berzins, R. Yeh, A prototyping language for real time software, IEEE Transactions on Software Engineering, October 1988, Vol. 14, No. 10, pp. 1409-1423
5. R. Steigerwald, Luqi, J. McDowell, A CASE tool for reusable software component storage and retrieval in rapid prototyping, Information and Software Technology, England, Vol. 38, No. 9, Nov. 1991, pp. 698-706
6. Luqi, V. Berzins, M. Shing and N. Nada, Evolutionary computer aided prototyping system (CAPS), to appear in the Proceedings of the TOOLS USA 2000 Conference, Santa Barbara, CA, July 30 - August 3, 2000

7. Jini technology architectural overview, URL: http://www.sun.com/jini/whitepapers/architecture.html, retrieved 07/31/2000.
8. V. Berzins, Luqi, B. C. Shultes, et al, Interoperability technology assessment for joint C4ISR systems, Technical Reports, Naval Postgraduate School, Monterey, CA, USA, October 1999
9. Luqi and M. Shing, real-time scheduling for Software prototyping, Journal of Systems Integration, Vol. 6, No. 1-2, pp. 41-72, 1996
10. Luqi, V. Berzins, M. Shing, N. Nada and C. Eagle, Computer aided prototyping system (CAPS) for heterogeneous systems development and Integration, to appear in the Proceedings of the 2000 Command and Control Research and Technology Symposium, Naval Postgraduate School, Monterey, CA, 26-28 June 2000
11. Jiang Guo, Luqi, Toward automated retrieval for a software component repository, Proceedings of IEEE International Conference and Workshop on the Engineering of Computer Based Systems (IEEE ECBS), Nashville, USA, March 7-12, 1999. Pp. 99-105

Subclassing errors, OOP, and practically checkable rules to prevent them

Oleg Kiselyov

Software Engineering, Naval Postgraduate School, Monterey, CA 93943

oleg@pobox.com, oleg@acm.org

Abstract

This paper considers an example of Object-Oriented Programming (OOP) leading to subtle errors that break separation of interface and implementations. A comprehensive principle that guards against such errors is undecidable. The paper introduces a set of *mechanically verifiable* rules that prevent these insidious problems. Although the rules seem restrictive, they are powerful and expressive, as we show on several familiar examples. The rules contradict both the spirit and the letter of the OOP. The present examples as well as available theoretical and experimental results pose a question if OOP is conducive to software development at all.

Keywords: object-oriented programming, subtyping, subclassing, implementation inheritance, C++, functional programming

1   Introduction

Decoupling of abstraction from implementation is one of the holy grails of good design. Object-oriented programming is claimed to be conducive to such a separation, and therefore to more reliable code. In the end, productivity and quality are the only true merits a programming methodology is to be judged upon. This article will discuss a simple example that questions if Object-Oriented Programming (OOP) indeed helps separate interface from implementation. First we demonstrate how easily subclassing errors arise and how difficult (in general, undecidable) it is to prevent them. We later introduce a set of expressive rules that preclude the subclassing errors, and can be mechanically verified. Incidentally the rules run contrary to the OOP precepts.

We take a rather familiar example that illustrates the difference between subclassing and subtyping: the example of Sets and Bags. The example is isomorphic to that of circles vs. ellipses or squares vs. rectangles. Section 2 introduces the example and carries it one step further, to a rather unsettling result: a "transparent" change in an implementation suddenly breaks client code that was written according to public interfaces. We set out to follow good software engineering practices; this makes the resulting failure even more ominous. Section 3 brings up a subclassing vs. subtyping dichotomy and the Liskov principle of behavioral substitutability. We show that Sets and Bags viewed as mutable or immutable *objects* are not subtypes of each other. The indiscriminate use of implementation inheritance indeed prevents separation of interface and implementation. In Section 4 we take a contrary point of view, of bags and sets as values without a hidden state and whose responses to external messages cannot be overridden. We prove that a set truly *is-a* bag; a set *is* substitutable for a bag, a set can always be manipulated as a bag, a set maintains every invariant of a bag – and it is still a set. The section also shows that if we abide by practically checkable rules we obtain a guarantee that the subtle subclassing errors cannot occur in principle. We will also show that the rules do not diminish the power of a language.

Inheritance and encapsulation, two staples of OOP, make checking of the Liskov Substitution Principle (LSP) for derived objects generally undecidable. On the other hand, the proposed rules, which *can* be checked at compile time, make derived values satisfy LSP.

The article aims to give a more-or-less "real" example, which we can run and see the result for ourselves. By necessity the example had to be implemented in some language. The present article uses C++. It appears however that similar code and similar conclusions can be carried on in many other object-oriented languages (e.g., Java, Python, etc).

2   Coupling of interface and implementation

Suppose I was given a task to implement a Bag – an unordered collection of possibly duplicate items (integers in this example). I chose the following interface:

```
typedef int const * CollIterator;      // Primitive but will do
class CBag {
 public:
  int size(void) const;
  int count(const int elem) const;
  virtual void put(const int elem);
  virtual bool del(const int elem);
  CollIterator begin(void) const;
  CollIterator end(void) const;

  CBag(void);
  virtual CBag * clone(void) const;
 private: ...          // implementation details elided
};
```

The class CBag defines usual methods to determine the number of all elements in a bag, to count the number of occurrences of a specific element, to put a new element into a bag and to remove one. The latter function returns false if the element to delete did not exist. We also define the standard enumerator interface [11] – methods begin() and end() – and a method to make a copy of the bag. Other operations of the CBag package are implemented without the knowledge of CBag's internals: the print-on operator <<, the union (merge) operator +=, and operators to compare CBags and to determine their structural equivalence. These functions use only the public interface of the CBag class:

```
void operator += (CBag& to, const CBag& from);
bool operator <= (const CBag& a, const CBag& b);
inline bool operator >= (const CBag& a, const CBag& b)
{ return b <= a; }
inline bool operator == (const CBag& a, const CBag& b)
{ return a <= b && a >= b; }
```

The complete code of the whole example is available in [7]. It has to be stressed that the package was designed to minimize the number of functions that need to know details of CBag's implementation. Following good practice, I wrote validation code (file vCBag.cc [7]) that tests all the functions and methods of the CBag package and verifies common invariants.

Suppose you are tasked with implementing a Set package. Your boss defined a set as an unordered collection where each element has a single occurrence. In fact, your boss even said that a set is a bag with no duplicates. You have found my CBag package and realized that it can be used with few additional changes. The definition of a Set as a Bag, with some constraints, made the decision to reuse the CBag code even easier.

```
class CSet : public CBag {
 public:
  bool memberof(const int elem) const
  { return count(elem) > 0; }

  // Overriding of CBag::put
  void put(const int elem)
  { if(!memberof(elem)) CBag::put(elem); }

  CSet * clone(void) const
  { CSet * new_set = new CSet();
    *new_set += *this; return new_set; }
  CSet(void) {}
};
```

The definition of a CSet makes it possible to mix CSets and CBags, as in set += bag; or bag += set; These operations are well-defined, keeping in mind that a set is a bag that happens to have the count of all members exactly one. For example, set += bag; adds all elements from a bag to a set, unless they are already present. On the other hand, bag += set; is no different than merging a bag with any other bag. You too wrote a validation suite to test all CSet methods (newly defined as well as inherited from a bag) and to verify common expected properties, e.g., a+=a ≡ a.

In my package, I have defined and implemented a function that, given three bags a, b, and c, decides if a+b is a subbag of c:

```
bool foo(const CBag& a, const CBag& b, const CBag& c)
{                       // Clone a to avoid clobbering it
  CBag & ab = *(a.clone());
  ab += b;              // ab is now the union of a and b
  bool result = ab <= c;
  delete &ab;
  return result;
}
```

It was verified in the test suite. You have tried this function on sets, and found it satisfactory.

Later on, I revisited my code and found my implementation of foo() inefficient. Memory for the ab object is unnecessarily allocated on heap. I rewrote the function as

```
bool foo(const CBag& a, const CBag& b, const CBag& c)
{
  CBag ab;
  ab += a;             // Clone a to avoid clobbering it
  ab += b;             // ab is now the union of a and b
  bool result = ab <= c;
  return result;
}
```

It has exactly the same interface as the original foo(). The code hardly changed. The behavior of the new implementation is also the same – as far as I and the package CBag are concerned. Remember, I have no idea that you are re-using my package. I re-ran the validation test suite with the new foo(): everything tested fine.

However, when you run your code with the new implementation of foo(), you notice that something *has* changed! The complete source code [7] contains tests that make this point obvious: Commands make vCBag1 and make vCBag2 run validation tests with the first and the second implementations of foo(). Both tests complete successfully, with the identical results. Commands make vCSet1 and make vCSet2 test the CSet package. The tests – other than those of foo() – all succeed. Function foo() however yields markedly different results. It is debatable which implementation of foo() gives truer results for CSets. In any case, changing internal algorithms of a pure function foo() while keeping the same interfaces is not supposed to break your code. What happened?

What makes this problem more unsettling is that both you and I tried to do everything by the book. We wrote a safe, typechecked code. We eschewed casts. g++ (2.95.2) compiler with flags -W and -Wall issued not a single warning. Normally these flags cause g++ to become very annoying. You did not try to override methods of CBag to deliberately break the CBag package. You attempted to preserve CBag's invariants (weakening a few as needed). Real-life classes usually have far more obscure algebraic properties. We both wrote validation tests for our implementations of a CBag and a CSet, and they passed. And yet, despite all my efforts to separate interface and implementation, I failed. Should a programming language or the methodology take at least a part of the blame? [10, 4, 1]

3 Subtyping vs. Subclassing

The breach of separation between CBag's implementation and interface is caused by CSet design's violating the Liskov Substitution Principle (LSP) [9]. CSet has been declared a subclass of CBag. Therefore, C++ compiler's typechecker permits passing a CSet object or a CSet reference to a function that expects a CBag object or reference. However, it is well known [3] that a CSet is not a *subtype* of a CBag. The next few paragraphs give a simple proof of this fact, for the sake of reference.

The previous section considered bags and sets from the OOP perspective – as objects that encapsulate state and behavior. Behavior means an object can accept a message, send a reply and possibly change its state. From this point of view, bags and sets are not subtypes of each other. Indeed, let us define a Bag as an object that accepts two messages: (send a-Bag 'put x) puts an element x into the Bag, and (send a-Bag 'count x) gives the occurrence count for x in the Bag (without changing a-Bag's state). Likewise, a Set is defined as an object that accepts two messages: (send a-Set 'put x) puts an element x into a-Set unless it was already there, (send a-Set 'count x) gives the count of occurrences of x in a-Set (which is always either 0 or 1). Throughout this section we use a different, concise notation to emphasize the general nature of the argument.

Let us consider a function

```
(define (fnb bag) (send bag 'put 5) (send bag 'put 5) (send bag 'count 5))
```

The behavior of this function, its contract, can be summed as follows: given a Bag, the function adds two elements into it and returns `(+ 2 (send orig-bag 'count 5))`. Technically you can pass to `fnb` a Set object as well. Just as a Bag, a Set object accepts messages `'put` and `'count`. However applying `fnb` to a Set object will break the function's post-condition stated above. Therefore, passing a set object where a bag was expected changes the behavior of a program. According to the LSP, a Set is not substitutable for a Bag – a Set cannot be a subtype of a Bag.

Let us consider another function

```
(define (fns set) (send set 'put 5) (send set 'count 5))
```

The behavior of this function is: given a Set, the function adds an element into it and returns 1. If you pass to this function a bag (which – just as a set – replies to messages `'put` and `'count`), the function `fns` may return a number greater than 1. This will break `fns`'s contract, which promised always to return 1.

One may claim that "A Set is not a Bag, but an ImmutableSet is an ImmutableBag." This is not correct either. An immutability per se does not confer subtyping to "derived" classes of data, as a variation of the previous argument shows [8]. C++ objects are record-based. Subclassing is a way of extending records, with possibly altering some slots in the parent record. Those slots must be designated as modifiable by a keyword `virtual`. In this context, prohibiting mutation and overriding makes subclassing imply subtyping. This is the reasoning behind BRules introduced below. However merely declaring the state of an object immutable is not enough to guarantee that derivation leads to subtyping: An object can override parent's behavior without altering the parent. This is easy to do when an object is implemented as a functional closure, when a handler for an incoming message is located with the help of some kind of reflexive facilities, or in prototype-based OO systems [8]. Incidently, if we do permit a derived object to alter its base object, we implicitly allow behavior overriding. For example, an object A can react to a message M by forwarding the message to an object B stored in A's slot. If an object C derived from A alters that slot it hence overrides A's behavior with respect to M.

The OOP point of view thus leads to a conclusion that neither a Bag nor a Set are subtypes of the other. The interface or an implementation of a Bag and a Set appear to invite subclassing of a Set from a Bag, or vice versa. Doing so however will violate the LSP – and we have to brace for strikingly subtle errors. The previous section intentionally broke the LSP to demonstrate how insidious the errors are and how difficult it may be to find them. Sets and Bags are very simple types, far simpler than the ones that typically appear in a production code. Since LSP when considered from an OOP point of view is undecidable, we cannot count on a compiler for help in pointing out an error. As Section 2 showed, we cannot rely on validation tests either. We have to *see* the problem [4, 10, 1].

4 Mechanically preventing subclassing errors

Bags and sets – as objects – indeed are not subtypes. Subclassing them violates LSP, which leads to insidious errors. Bags and sets however do not have to be viewed as objects. We can take them as pure values, without any state or intrinsic behavior – just like the numbers are. In Section 2, CBag and CSet objects encapsulated a hidden state – a collection of integers. The objects had an ability to react to messages, e.g., `put` and `del`, by altering their state. In this section we re-do the example of Section 2 using a different approach. Bags and sets no longer have a state that is distinct from their identity and that can be altered. Equally important we do not allow any changes to the behavior of bags and sets with respect to applicable operations, by overriding or otherwise. In other words, every post-condition of a bag or a set constructor holds throughout the lifespan of the constructed values. This approach makes the subclassing problems and breach of encapsulation disappear. It turns out that a set truly *is-a* bag; a set is substitutable for a bag, a set can always be manipulated as a bag, a set maintains every invariant of a bag – and it is still a set.

The LSP says, "If for each object o1 of type S there is another object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2, then S is a subtype of T." If type T denotes a set of values that carry their own behavior, and if values of type S can override some of T values behavior, the LSP is undecidable. Indeed, a mechanical application of LSP must at least be able to verify that all methods overridden in S terminate whenever the corresponding methods in T terminate. This is generally impossible. On the other hand, if T denotes a set of (structured) data values, and S is a subset of these values – e.g., restricted by range, parity, etc. – the LSP is trivially satisfied.

This section also shows that if one abides by mechanically verifiable rules he obtains a guarantee that the subtle subclassing errors cannot occur in principle. The rules do not reduce the power of a language.

4.1 BRules

Suppose I was given a task to implement a Bag – an unordered collection of possibly duplicate items (integers in this example). This time my boss laid out the rules, which we will refer to as *BRules*:

- no virtual methods or virtual inheritance
- no visible members or methods in any public data structure (that is, in any class declared in an .h file)
- no mutations to public data structures
    - a strict form: no assignments or mutations whatsoever
    - a less strict form: no function may alter, directly or indirectly, any data it receives as arguments

The rules break the major tenets of OOP: for example, values no longer have a state that is separate from their identity. Prohibitions on virtual methods and on modifications of public objects are severe. It appears that not much of C++ is left. Surprisingly I still can implement my assignment without losing expressiveness – and perhaps even gaining some. The exercise will also illustrate that C++ does indeed have a pure functional subset [12], and that you can program in C++ without assignments.

4.2 Interface and implementation of a FBag

```
class FBag {
 public:
  FBag(void);
  FBag(const FBag& another);     // Copy-constructor
  ~FBag(void);

 private:
  class Cell;                    // Opaque type
  const Cell * const head;
  FBag(const Cell * const cell); // Private constructor
                   // Declaration of three friends elided
};
```

Indeed, there are no virtual functions, no methods or public members. We also declare functions that take a FBag as (one of the) arguments and return the count of all elements or a specific element in the bag, print the bag, *fold* [5] over the bag, compare two bags for structural equivalence, verify bag's invariants, merge two bags, add or delete an element. The latter three functions do not modify their arguments; they return a new FBag as their result. It must be stressed that the functions that operate on a FBag are not FBag's methods; in particular, they are not a part of the class FBag, they are not inherited and they cannot be overridden. The implementation is also written in a functional style. FBag's elements are held in a linked list of cells, which are allocated from a pre-defined pool. The pool implements a mark-and-sweep garbage collection, in C++.

Forgoing assignments does not reduce expressiveness as the following snippet from the FBag code shows; the snippet implements the union of two FBags:

```
struct union_f {
  FBag operator() (const int elem, const FBag seed) const {
    return put(seed,elem);
  }
};
FBag operator + (const FBag& bag1, const FBag& bag2)
{
  return fold(bag1,union_f(),bag2);
}
```

Following good practice, I wrote a validation code (file vFBag.cc [7]) that tests all the functions of the FBag package and verifies common invariants.

4.3 Implementation of a FSet. FSet is a subtype of a FBag

Suppose you are tasked with implementing a Set package. Your boss defined a set as an unordered collection where each element has a single occurrence. In fact, your boss even said that a set is a bag with no duplicates. You have found my FBag package and realized that it can be used with few additional changes. The definition of a Set as a Bag (with some constraints) made the decision to reuse the FBag code even easier.

56

```
class FSet : public FBag {
 public:
  FSet(void) {}
  FSet(const FBag& bag) : FBag(remove_duplicates(bag)) {}
};

bool memberof(const FSet& set, const int elem)
{ return count(set,elem) > 0; }
```

Surprisingly, this is the *whole* implementation of a FSet. A set is fully a bag. Because FSet constructors eventually call FBag constructors and do no alter the latter's result, every post-condition of a FSet constructor implies a post-condition of a FBag constructor. Since FBag and FSet values are immutable, the post-conditions that hold at their birth remain true through their lifespan. Because all FSet values are created by an FBag constructor, all FBag operations automatically apply to an FSet value. This concludes the proof that an FSet is a *subtype* of a FBag.

The FBag.cc package [7] has a function verify(const FBag&) that checks to make sure its argument is indeed a bag. The function tests FBag's invariants, for example:

```
const FBag bagnew = put(put(bag,5),5);
assert( count(bagnew,5) == 2 + count(bag,5) &&
         size(bagnew) == 2 + size(bag) );
assert( count(del(bagnew,5),5) == 1 + count(bag,5) );
```

Your validation code passes a non-empty set to this function to verify the set is indeed a bag. You can run the validation code vFSet.cc [7] to see for yourself that the test passes. On the other hand, FSets do behave like Sets:

```
const FSet a112 = put(put(put(FSet(),1),1),2);
assert( count(a112,1) == 1 );

const FSet donce = FSet() + a112;
const FSet dtwice = donce + a112;
assert( dtwice == a112 );
```

where a112 is a non-empty set. The validation code vFSet.cc you wrote contains many more tests like the above. The code shows that a FSet is able to pass all of FBag's tests as well as its own. The implementation of FSets makes it possible to take a union of a set and a bag; the result is always a bag, which can be made a set if desired. There are corresponding test cases as well.

To clarify how an FSet may be an FBag at the same time, let us consider one example in more detail:

```
       // An illustration that an FSet is an FBag
int cntb(const FBag v) {
  FBag b1 = put(v, 5);  FBag b2 = put(b1, 5);
  FBag b3 = del(b2, 5);
  return count(b3, 5); }
const int cb1 = cntb(FBag()); // cb1 has the value of 1
const int cb2 = cntb(FSet()); // cb2 has the value of 1

       // An illustration that an FSet does act as a set
int cnts(const FSet v) {
  FSet s1 = put(v, 5); FSet s2 = put(s1, 5);
  FSet s3 = del(s2, 5);
  return count(s3, 5); }
const int cs = cnts(FSet()); // cs has the value of 0
```

This example is one of the test cases in vFSet.cc [7]. You can run it and check the results for yourself. Yet it is puzzling: how come cs has the value different from that of cb1 if there is no custom del() function for FSets? The statement FSet s2 = put(s1, 5); is the most illuminating. On the right-hand side is an expression: putting an element 5 to a FBag/FSet that already has this element in it. The result of that expression is a FBag {5,5}, with two instances of element 5. The statement then constructs a FSet s2 from that bag. A FSet constructor is invoked. The constructor takes the bag {5,5}, removes the duplicate element 5 from it, and "blesses" the resulting FBag to be a FSet as well. Thus s2 will be a FBag and a FSet, with one instance of element 5. In fact, s1 and s2 are identical. A FSet constructor guarantees that a FBag it constructs contains no duplicates. As objects are immutable, this invariant holds forever.

57

4.4 Discussion

Surprising as it may be, assertions "a Set is a Bag with no duplicates" and "a Set always acts as a Bag" do not contradict each other, as the following two examples illustrate:

Let {value ...} be an unordered collection of values: a Bag. Let us consider the following values: $vA : 42$, $vB : \{42\}$, $vC : \{43\}$, $vD : \{42\,43\}$, $vE : \{42\,43\,42\}$ vA is not a collection; vB, vC, vD, and vE are bags. vB, vC, and vD are also Sets: unordered collections without duplicates. vE is not a Set. Every Set is a Bag but not every Bag is a Set.	Let *uf-integer* denote a natural number whose prime factors are unique. Let us consider the following values: $vA : \frac{5}{4}$, $vB : 42$, $vC : 43$, $vD : 1806$, $vE : 75852$ vA is not an integer; vB, vC, vD, and vE are integers. vB, vC, and vD are also uf-integers. vE is not a uf-integer as it is a product $2*2*3*3*7*7*43$ with factors 2, 3, and 7 occurring several times. Every uf-integer is an integer but not every integer is a uf-integer.
We introduce operations *merge* (infix +) and *subtract* (infix −). Both operations take two Bags and return a Bag. Either of the operands, or both, may also be a Set. The result, a Bag, may or may not be a Set. For example, $vB + vC \Rightarrow vD$ Both of the operands and the result are also Sets $vB + vD \Rightarrow vE$ The argument Bags are also Sets, but the resulting Bag is not a Set $vE + vE \Rightarrow \{42\,43\,42\,42\,43\,42\}$ None of the Bags here are Sets $vD - vC \Rightarrow vB$ The argument Bags are also Sets, so is the result. $vE - vC \Rightarrow \{42\,42\}$ One of the arguments is a Set, the resulting Bag is not a Set. $vE - vE \Rightarrow \{\}$ The argument Bags are not Sets, but the resulting Bag is.	We introduce operations *multiply* (infix *) and *reduce* (infix %): $a\%b = a/gcd(a, b)$. Both operations take two integers and return an integer. Either of the operands, or both, may also be a uf-integer. The result, an integer, may or may not be a uf-integer. For example, $vB * vC \Rightarrow vD$ Both of the operands and the result are also uf-integers $vB * vD \Rightarrow vE$ The argument integers are also uf-integers, but the resulting integer is not a uf-integer $vE * vE \Rightarrow 5753525904$ None of the integers here are uf-integers $vD\%vC \Rightarrow vB$ The argument integers are also uf-integers, so is the result $vE\%vC \Rightarrow 1764$ One of the arguments is a uf-integer, the resulting integer is not a uf-integer $vE\%vE \Rightarrow 1$ The argument integers are not uf-integers, but the resulting integer is.

Bags are closed under operation *merge* but subsets of Bags – Sets – are not *not* closed under *merge*. On the other hand, both Bags and Sets are closed under *subtract*.	Integers are closed under operation *multiply* but subsets of integers – uf-integers – are *not* closed under *multiply*. On the other hand, both integers and uf-integers are closed under *reduce*.
We may wish for a merge-like operation that, being applied to Sets, always yields a Set. We can introduce a new operation: $merge - if - not - there$. We can define it specifically for Sets. Alternatively, the operation can be defined on Bags; it would apply to Sets by the virtue of inclusion polymorphism as every Set is a Bag. Sets *are* closed with respect to $merge - if - not - there$. On the other hand, to achieve closure of Sets under *merge* we can project – coerce – the result of merging of two Sets back into Sets, a subset of Bags. The FBag/FSet package took this approach. If we *merge* two FSets and want to get an FSet in result we have to specifically say so, by applying a projection (coercion) operator: `FSet::FSet(const FBag& bag)`. That operator creates a new FBag without duplicates. This fact makes the latter a FSet. Thus $FSet(vB + vD) \Rightarrow vD$, an FSet.	We may wish for a multiply-like operation that, being applied to uf-integers, always yields a uf-integer. We can introduce a new operation: lcm, the least common multiple. This operation is well-defined on integers; it would apply to uf-integers by the virtue of inclusion polymorphism as every uf-integer is an integer. uf-integers *are* closed with respect to the lcm operation. On the other hand, to achieve closure of uf-integers under *multiply* we can project – coerce – the product of two uf-integers back into uf-integers, a subset of integers. If we *multiply* two uf-integers and want to get a uf-integer in result we have to specifically say so, by applying a projection (coercion) operator: $remove - duplicate - factors$. That operator creates a new integer without duplicate factors. This fact makes the resulting integer a uf-integer. Thus $uf - integer(vB * vD) \Rightarrow vD$, a uf-integer

It has to be stressed that the two columns of the above table are not merely similar: they are isomorphic. Indeed, the right column is derived from the left column by the following substitution of words that preserves meaning: Bag ↔ integer, Set ↔ uf-integer, merge ↔ multiply, subtract ↔ reduce. The right column sounds more "natural" – so should the left column as integers and uf-integers are representations for resp. FBags and FSets.

From an extensional point of view [2], a type denotes a set of values. By definition of a FSet, it is a particular kind of FBag. Therefore, a set of all FSets is a subset of all FBags: FSet is a subtype of FBag. A FBag or a FSet do not have any "embedded" behavior – just as integers do not have an embedded behavior. Behavior of numbers is defined by operations, mapping from numbers to numbers. Any function that claims to accept every member of a set of values identified by a type T will also accept any value in a subset of T. Frequently a value can participate in several sets of operations: a value can have several types at the same time. For example, a collection { 42 } is both a Bag and a Set. This fact should not be surprising. In C++, a value typically denoted by a numeral 0 can be considered to have a character type, an integer type, a float type, a complex number type, or a pointer type, for any declared or yet to be declared pointer type. This lack of behavior is what puts FBag and FSet apart from CBag and CSet discussed in the previous article. FSet is indeed a subtype of FBag, while CSet is not a subtype of a CBag as CSet has a different behavior. Incidentally LSP is trivially satisfied for values that do not carry their own behavior. FBags and FSets are close to so-called predicate classes. Since instances of FSets are immutable, the predicate needs to be checked only at a value construction time.

4.5 Polymorphic programming with BRules

The FSet/FBag example above showed BRules in the context of subtypes formed by a restriction on a base type. As it turns out, BRules work equally well with existential (abstract) types. To illustrate this point, the source code accompanying this article [7] contains three implementations of a collection of polymorphic values. The collection is populated by Rectangles and Ellipses, which are instances of concrete classes implementing a common abstract base class Shape. A Shape is an existential type that knows how to draw, move and resize itself. A file Shapes-oop.cc gives the conventional, OOP-like implementation, with virtual functions and such. A file Shapes-no-oop.cc is another implementation, also in C++. The latter follows BRules, has no assignments or virtual functions. Any particular Shape value is created by a Shape constructor and is not altered after that. Shapes-no-oop.cc achieves polymorphic programming with the full separation of interface and implementation: If an implementation of a concrete Shape is changed, the code that constructs and uses Shapes does not even have to be recompiled! The file defines two concrete instances of the Shape: a Square and a Rectangle. The absence of mutations and virtual functions guarantees that any post-condition of a Square or a Rectangle constructor implies the post-condition of a Shape. Both particular

shapes can be passed to a function `set_dim(const Shape& shape, const float width, const float height);` Depending on the new dimensions, a square can *become* a rectangle or a rectangle square. You can compile Shapes-no-oop.cc and run it to see that fact for yourself.

It is instructive to compare Shapes-no-oop.cc with Shapes-h.hs, which implements the same problem in a purely functional, strongly-typed language Haskell. All three code files in the Shapes directory solve the same problem the same way. Two C++ code files – Shapes-oop.cc and Shapes-no-oop.cc – look rather different. On the other hand, the purely functional Shapes-no-oop.cc and the Haskell code Shapes-h.hs are uncanny similar – in some places, frighteningly similar. This exercise shows that BRules do not constrain the power of a language even when abstract data types are involved.

5 Conclusions

It is known, albeit not so well, that following the OOP letter and practice may lead to insidious errors [10, 1]. Section 2 of this article showed how subtle the errors can be even in simple cases. In theory, there are rules – LSP – that could prevent the errors. Alas, the rules are in general undecidable and not *practically reinforceable.*

In contrast, BRules introduced in this article can be statically checked at compile time. The rules outlaw certain syntactic constructions (for example, assignments in some contexts, and non-private methods) and keywords (e.g., `virtual`). It is quite straightforward to write a lint-like application that scans source code files and reports if they conform to the rules. When BRules are in effect, subtle subclassing errors like the ones shown in Section 2 become impossible. To be more precise, with BRules, *subclassing implies subtyping*. Subclassing by definition is a way of creating (derived) values by extending, restricting, or otherwise specializing other, parent values. A derived value constructor must invoke a parent value constructor to produce the parent value. The former constructor often has a chance to alter the parent constructor's result before it is cast or incorporated into the derived value. If this chance is taken away, the post-condition of a derived value constructor implies the post-condition of the parent value. Disallowing any further mutations guarantees the behavioral substitutability of derived values for parent values at all times.

As the examples in this article showed, following BRules does not diminish the power of the language. We can still benefit from polymorphism, we can still develop practically relevant code. Yet BRules blur the distinction between the identity and the state, a characteristic of objects. BRules are at odds with the practice if not the very mentality of OOP. This begs the question: Is OOP indeed conducive to software development?

One can argue that OOP – as every powerful technique – requires extreme care: knives are sharp. Likewise, goto is expressive, and assembler- or microcode-level programming are very efficient. All of them can lead to bugs that are very difficult, statically impossible, to find. On the other hand, if you program, for example, in Scheme, you never have to deal with an "invalid opcode" exception. That error becomes simply impossible. Furthermore, "while opinions concerning the benefits of OOSD [Object-Oriented Software Development] abound in OO literature, there is little empirical proof of its superiority" [6].

References

[1] Cardelli, L. Bad Engineering Properties of Object-Oriented Languages. ACM Comp. Surveys 28(4es), 1996, article 150.

[2] Cardelli, L., Wegner, P. On Understanding Types, Data Abstraction, and Polymorphism. ACM Comp. Surveys, 17(4): December 1985, pp. 471-522.

[3] Cook, W.R., Hill, W.L., Canning, P.S. Inheritance Is Not Subtyping. In: Carl A. Gunter and John C. Mitchell. Theoretical Aspects of Object- Oriented Programming. MIT Press. ISBN 0-262-07155-X.

[4] Hatton, L. Does OO sync with how we think? IEEE Software 15(3) , May-June 1998, pp. 46 -54.

[5] Hutton, G. A tutorial on the universality and expressiveness of fold. Journal of Functional Programming, 9(4):355-372, July 1999.

[6] Johnson, R.A. The Ups and Downs of Object-Oriented Systems Development. Comm. ACM 43(10), October 2000, pp. 69-73.

[7] Kiselyov, O. Complete code that accompanies the article. <http://pobox.com/~oleg/ftp/packages/subclassing-problem.tar.gz>, August 4, 2000.

[8] Kiselyov, O. Subtyping, Subclassing, and Trouble with OOP. <http://pobox.com/~oleg/ftp/Computation/Subtyping/index.html>, August 4, 2000.

[9] Liskov, B., Wing, J. M. A Behavioral Notion of Subtyping. ACM Trans. Programming Languages and Systems, 16(6), November 1994, pp. 1811-1841.

[10] Ousterhout, J.K. Scripting: Higher-Level Programming for the 21st Century. IEEE Computer, March 1998, pp. 23-30.

[11] Standard Template Library Programmer's Guide. SGI Inc, 1996-1999. <http://www.sgi.com/tech/stl/>.

[12] Stroustrup, B. The Real Stroustrup Interview. IEEE Computer 31(6), June 1998, pp. 110-114.

The Use of Computer Aided Prototyping for Re-engineering Legacy Software[1]

Luqi, V. Berzins, M. Shing
Department of Computer Science
Naval Postgraduate School
Monterey, CA 93943

Abstract

Re-engineering is typically needed when a system performing a valuable service must change, and its current implementation can no longer support cost-effective changes. The process of re-engineering old procedural software to a modern object-oriented architecture introduces certain complexities into the software analysis process. The direct products of reverse engineering, such as requirements or design specifications, are likely to have a functionally based structure. As a result, some transformation of the recovered requirements and design specifications is necessary in order to obtain specifications for the new structures. It is often very difficult to quickly determine if the transformed specification is a true representation of the desired requirements. This paper discusses the effective use of computer-aided prototyping techniques for re-engineering legacy software, and presents results of a case study which showed that prototyping can be a valuable aid in re-engineering of legacy systems, particularly in cases where radical changes to system conceptualization and software structure are needed.

Keywords: Software re-engineering, Object-oriented architecture, Computer-aided prototyping, Software evolution, Combat simulation

1. Introduction

Legacy systems embody substantial institutional knowledge, which includes basic and refined requirements, design decisions, and invaluable advice and suggestions from domain users that have been implemented over the years. To effectively use these assets, it is important to employ a systematic strategy for continued evolution of the current system to meet the ever-changing mission, technology and user needs. Re-engineering has frequently been proven to be more cost effective than new development and is also known to better promote continuous software evolution.

However, the institutional knowledge implicit in a legacy system is difficult to recover after many years of operation, evolution, and personnel change. These software systems were originally written twenty or more years ago using what many now view archaic and ad-hoc methods. Such legacy systems usually lack accurate documentation, modular structure, and coherent abstractions that correspond to current or projected requirements. Past optimizations and design changes have spread design decisions that now must be changed over large areas of the code, and may have introduced inconsistencies and faults.

Software re-engineering can be defined as the systematic transformation of an existing system into a new form to realize quality improvements, such as increased or enhanced functionality, better maintainability, configurability, reusability, performance, or evolvability at a reduced cost, schedule, or risk to the customer. This process involves recovering existing software artifacts from the system and then transforming and re-organizing them as a basis for future evolution of the system. Since typical legacy systems were originally designed and implemented using a functionally based approach, some transformation of the recovered information is necessary in order to obtain an object-oriented model. It is often very difficult to obtain a transformed specification that accurately represents the desired requirements.

Since legacy systems are usually re-engineered only when the existing systems need some kind of improvement, it is unlikely that the initial version of the reconstructed requirements adequately reflects

current user needs. Prototyping provides a means to identify and validate changes to system requirements while simultaneously enabling prospective users to get a feel for new aspects of the proposed system. It is a well-established approach that can be highly effective in increasing software quality [15]. When used in conjunction with conducting a major re-engineering effort, prototyping can be extremely useful in assisting in many areas of software modification, validation, risk reduction, and the refinement of new software architectures and user requirements.

This paper describes a case study that illustrates the effective use of computer-aided prototyping techniques for re-engineering legacy software [3, 16]. The case study consists of developing an object-oriented modular architecture for the existing US Army Janus(A) combat simulation system [19], and validating the architecture via an executable prototype using the Computer Aided Prototyping System (CAPS), a research tool developed at the Naval Postgraduate School [14]. Janus(A) is a software-based war game that simulates ground battles between up to six adversaries [9]. It is an interactive, closed, stochastic, ground combat simulation with color graphics. Janus is "interactive" in that command and control functions are entered by military analysts who decide what to do in crucial situations during simulated combat. The current version of Janus operates on a Hewlett Packard workstation and consists of over 350,000 lines of FORTRAN code. The FORTRAN modules are organized as a flat structure and interconnected with one another via 129 FORTRAN COMMON blocks, resulting in a software structure that makes modification to Janus very costly and error-prone. The Software Engineering group at the Naval Postgraduate School was tasked to extract the existing functionality through reverse engineering and to create a base-line object-oriented architecture that supports existing and required enhancements to Janus functionality.

The paper presents the re-architecturing process and the resultant object-oriented architecture in Sections 2. Section 3 describes the use of computer aided prototyping to validate the resultant architecture and Section 4 draws some conclusions.

2. The Re-Architecturing Process

The re-architecturing process used in the case study consists of 3 major phases: reverse engineering, object-oriented design and design validation via prototyping (Figure 1).
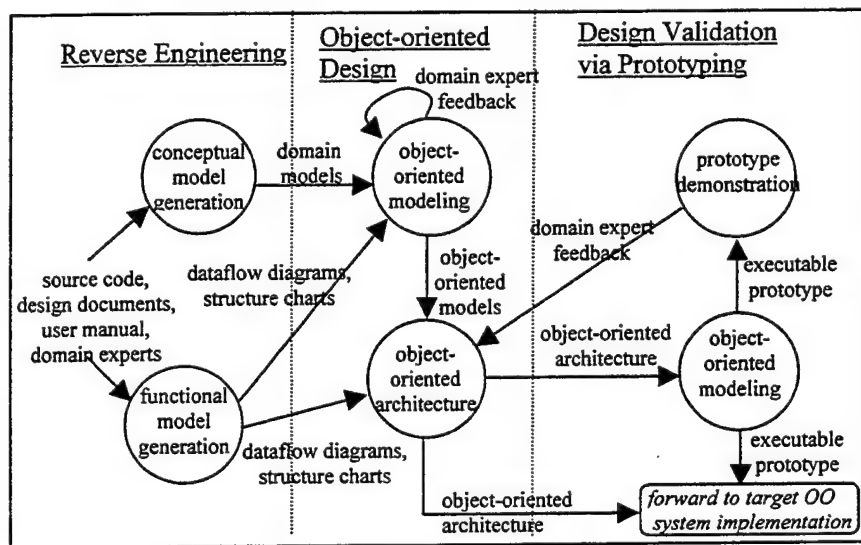
Figure 1. The object-oriented re-architecturing process.

63

2.1 Reverse Engineering

The first phase is reverse engineering. Input to this phase includes the legacy source code, design documents, user manuals, and information from domain experts. Since the goal of the initial re-engineering effort is to duplicate the functionality of the existing system within a modular, extensible architecture and to reuse domain concepts, models and algorithms instead of the existing code, we should avoid including any requirements/constraints that are consequences of issues related to FORTRAN implementation. The best places to extract domain concepts from the existing system are the user manuals and the database management system manuals. These manuals were written using the lingo of the user community and should be relatively free of implementation details. We found the JANUS Data Base Management Program Manual [10] particularly useful because it contains detailed information on what kind of data are needed to model the battlefield and how they are organized (logically) in the database. The top-level structure of the database is shown in Figure 2.
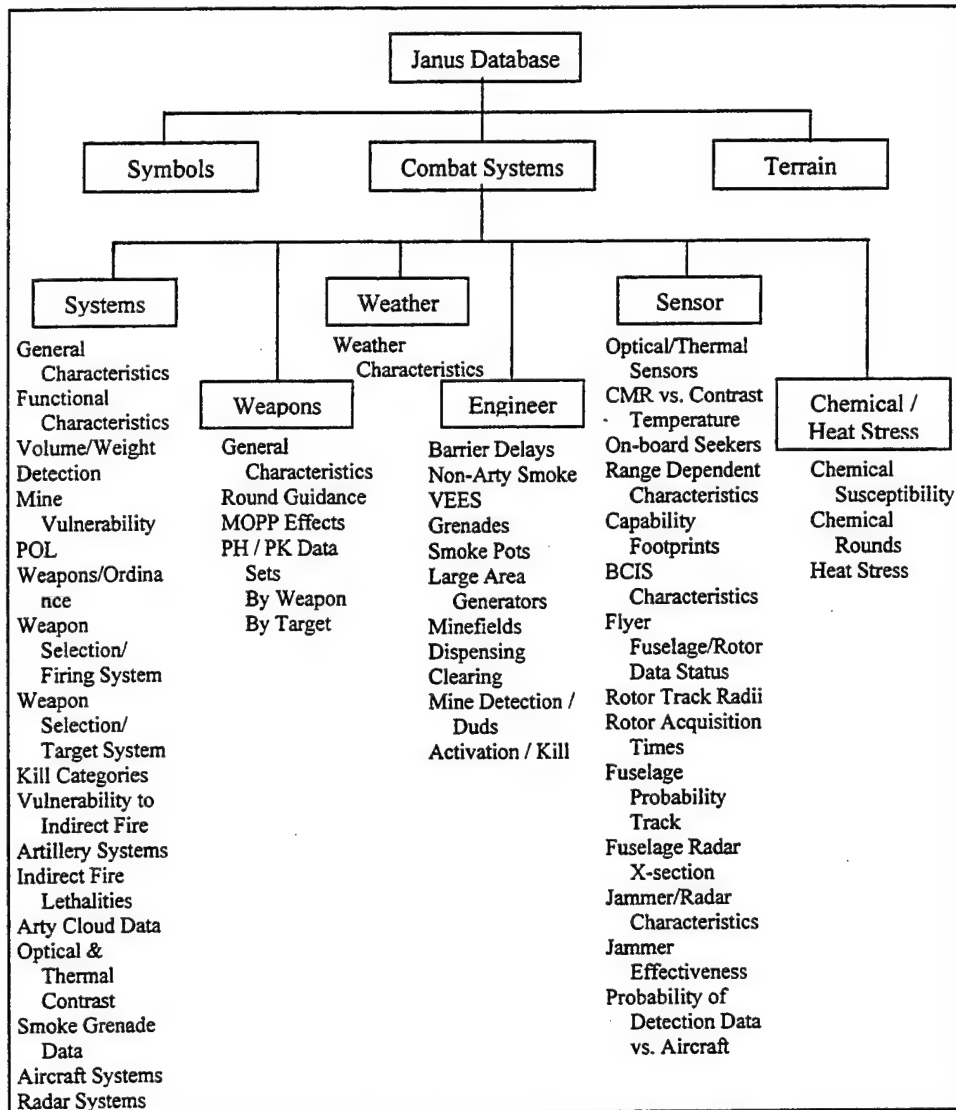
Figure 2. The top-level structure of the Janus Database.

Not shown in Figure 2 are the interdependencies between the data, whereby data entered in one category affect directly or indirectly the data in other categories. For example, the barrier delay attributes of the Engineer Data depend on specific weather conditions derived from the Weather Data and system functional characteristics derived from the System Data. The overall network of interdependencies is highly complex and can only be understood through construction and analysis of a functional model of the existing Janus software.

Analysis of the legacy implementation of 350,000 lines of source code is a daunting but inescapable part of this step. We recoiled from the magnitude of this effort and analyzed the Janus User's manual [9], the Janus Programmer's Manual [7], the Janus Software Design Manual [8], and the Janus Algorithm Document [18] instead. These documents helped us get started because they contained higher level information and were much shorter than the code. However, they were also older, and it was a constant struggle to determine which parts were still accurate, and which were not. In hindsight, avoiding analysis of the code was a mistake that slipped the schedule of the project by several months. Understanding a design of this complexity requires time for mental digestion, even with tool support and judicious sampling. We should have started analysis of the code right away and should have persistently continued this task in parallel with all other re-engineering activities. Cross-fertilization between all the tasks would have helped us recognize some dead-end directions earlier and would have enabled us to spend meeting time more effectively.

Using manual techniques augmented with the text matching tool *grep* [1], which takes a regular expression and a list of files and lists the lines of those files that match the pattern, we were able to walk through the code and get a fairly good idea of what each subroutine was designed to do. We also used the Software Programmers' Manual [7] to aid in understanding each subroutine's function. In doing so we were able to group the subroutines by functionality to get a better understanding of the major data flows between programs and develop functional models from the data flows. We used CAPS to assist in developing the abstract models [3]. CAPS allowed us to rapidly graph the gathered data and transform it into a more readable and usable format. Additionally, CAPS enabled us to concurrently develop our diagrams, and then join them together under the CAPS environment, where they can be used to generate an executable model.

We also had a series of brief meetings with the client, TRAC-Monterey, asking questions and making notes on the system's operation and its current functionality. We paid attention to the client's view of the system to gather their ideas on its strengths, weaknesses, and desired and undesired functionality. These meetings were indispensable because they gave us information that was not present in the code. Since we were not familiar with the domain of ground combat simulation, we were using these meetings to determine the requirements of this domain, often playing the role of "smart ignoramuses" [4]. Domain analysis has been identified as an effective technique for software re-engineering [17]. Our experience suggests that competent engineers unfamiliar with the application domain have an essential role in re-engineering as well as in requirements elicitation because lack of inessential information about the application domain makes it easier to find new, simpler design structures and architectural concepts to guide the re-engineering effort.

2.2. Object-Oriented Design

Next, we developed object models and architecture of the Janus System using the aforementioned materials and products, to create the modules and associations amongst them. Information modeling is needed to support effective re-engineering of complex systems [5]. This was probably the most difficult and most important phase. It required a great deal of analysis and focus to transform the currently scattered sets of data and functions into small, coherent and realizable objects, each with its own attributes and operations. In performing this phase, we used our knowledge of object-oriented analysis and the UML notations to create the classes and associated attributes and operations [20]. This was a crucial phase because we had to ensure that the classes we created accurately represented the functions and procedures currently in the software.

Restructuring software to identify data abstractions is a difficult part of the process. Transformations for meaning-preserving restructuring can be useful if tool support is available [6]. We used the HP-UNIX systems at the TRAC-Monterey facility to run the Janus simulation software to aid in verifying and supplementing the information we obtained from reviewing the source code and documentation. This step enabled us to better analyze the simulation system, gaining insight into its functionality and further concentrate on module definition and refinement.

The re-engineering team met several times each week for a period of two and a half months to discuss the object models for the Janus core data elements and the object-oriented architecture for the Janus System. We presented the findings to the Janus domain experts at least once per week to get feedback on the models and architectures being constructed. In addition, the re-engineering team also presented the findings to members of the OneSAF project, the Combat21 project, and the National Simulation Center project. We found that information from these domain experts was essential for understanding the system, particularly in cases where the legacy code did not correspond to stakeholder needs. This supports the hypothesis advanced in [11] that the involvement of domain experts is critical for nontrivial re-engineering tasks.

Early involvement of the stakeholders in the simulation community also paid off in the long run. Both the National Simulation Center and Combat21 projects were able to save time and money by reusing our work and came up with designs that look remarkably like ours (although much larger). Now, OneSAF developers have been directed to look at the Combat21 class design and reuse as much as possible. So, our efforts have directly benefited other simulation developers.

Based on the feedback from the domain experts, the re-engineering team revised the object models for the Janus core elements and developed a 3-tier object-oriented architecture for the Janus System (Figure 3).

Note: Lines showing the dependency of the Combat Systems DBMS, Scenario Management, JANUS Combat Simulation, JAAWS, POSTP, DB Utilities and Pass Interface subsystems on the Core Elements packages are omitted from the diagram to keep it clear and simple.
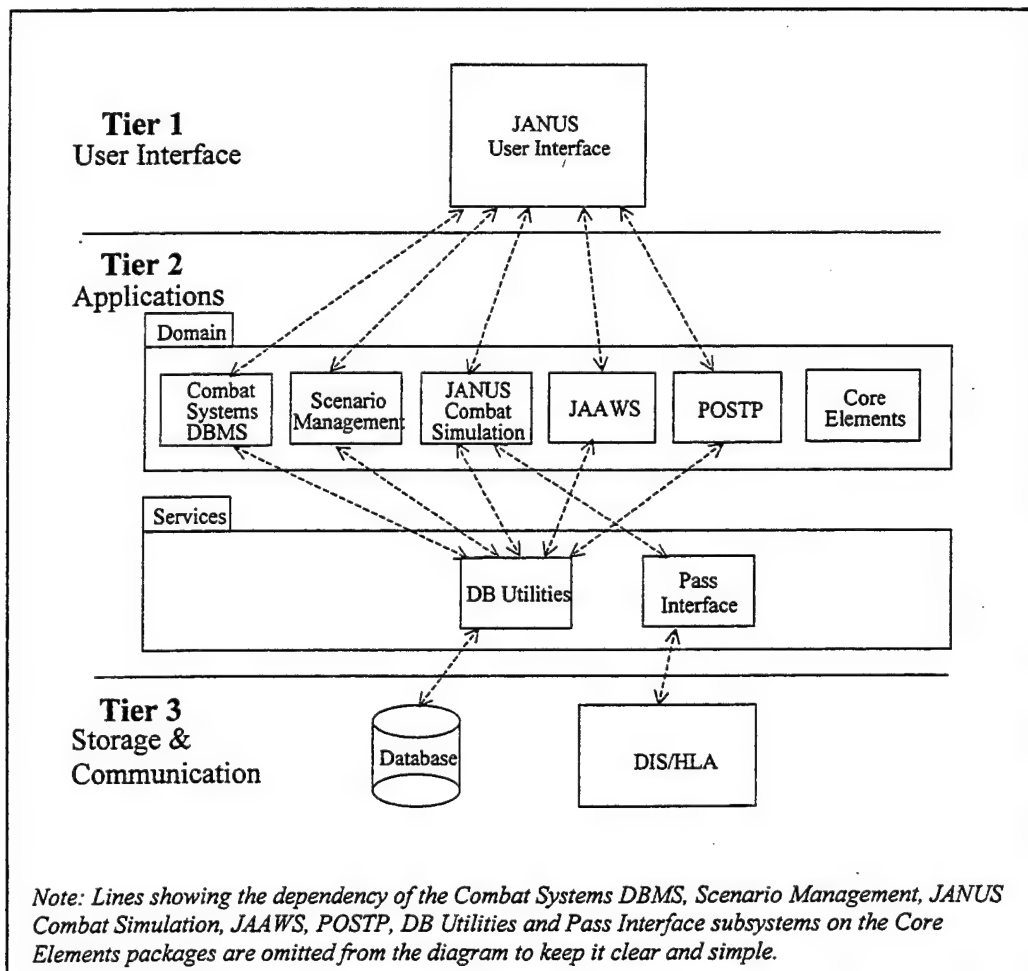
Figure 3. The resultant 3-tier object-oriented architecture.

We extracted most of the data and operations from the existing Combat System DBMS, Scenario Management, Janus Combat Simulation, JAAWS and POSTP subsystems and encapsulated them as simulation objects in the Core Elements package, leaving only application specific control codes that use the simulation objects in each of these five subsystems. Figures 4 and 5 show the top level class structures of the object models of the core elements. Details of the associated attributes and operations can be found in [3, 22] and are omitted from these diagrams due to space limitations.
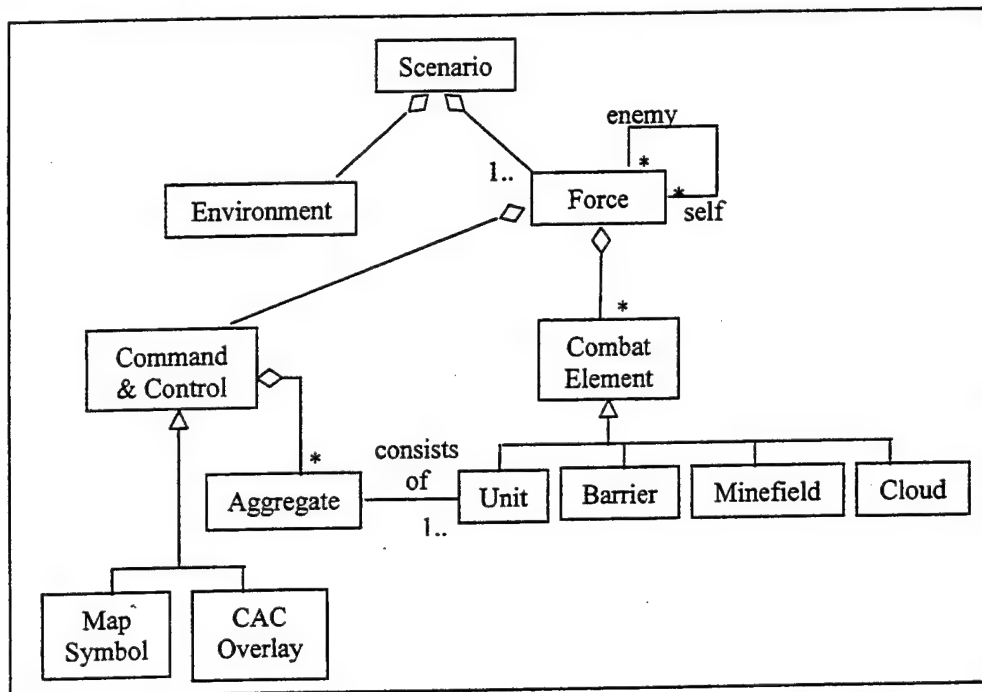
Figure 4. The top-level structure of the Janus Core Elements Object Model.
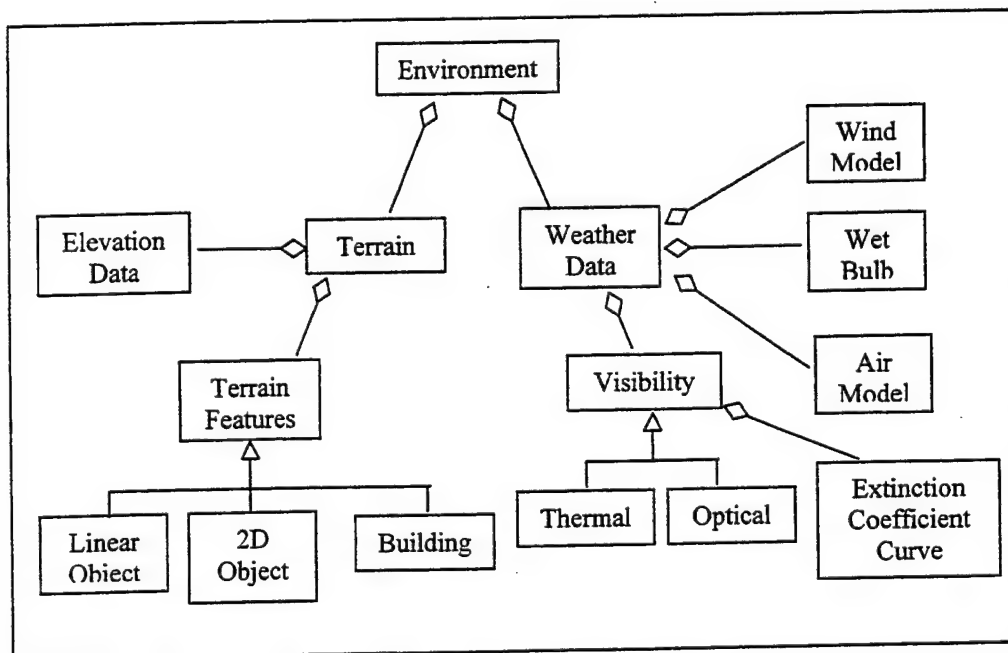
Figure 5. The Environment Object Class.

Central to the Janus Combat Simulation Subsystem is the program RUNJAN, which is the main event scheduler for the simulation. RUNJAN determines the next scheduled event and executes that event. If the next scheduled event is a simulation event, RUNJAN will advance the game clock to the scheduled time of the event and perform that event. The existing Janus Simulation System uses 17 different categories to characterize the events. RUNJAN then handles these 17 events using the event handlers shown in Figure 6.

```
1)   DOPLAN - Interactive Command and Control activities
2)   MOVEMENT - Update unit positions
3)   DOCLOUD - Create and update smoke and dust clouds
4)   STATEWT - Periodic activity to write unit status to disk
5)   RELOAD - Plan and execute the direct fire events
6)   INTACT - Update the graphics displays
7)   CNTRBAT - Detect artillery fire
8)   SEARCH - Update target acquisitions, choose weapons against potential targets, and
     schedule potential direct fire events
9)   DOCHEM - Create chemical clouds and transition units to different chemical states
10)  FIRING - Evaluate direct fire round impacting and execute indirect fire missions
11)  IMPACT - Evaluate and update the results of an indirect round impacting
12)  RADAR - Update an air defense radar state and schedule direct fire events for "normal"
     radar
13)  COPTER - Update helicopter states
14)  DOARTY - Schedule indirect fire missions
15)  DOHEAT - Update unit's heat status
16)  DOCKPT - Activity to record automatic checkpoints
17)  ENDJAN - Housekeeping activity to end the simulation
```

Figure 6. The event handlers for the legacy Janus system.

. Like all typical Fortran programs, the existing event scheduler uses global arrays and matrices to maintain the attributes of the objects in the simulation. Hence, one of the major tasks in designing an object-oriented architecture for the Janus Combat Simulation Subsystem was to distribute the event handling functions to individual objects. However, many of the current event handler categories contained redundant code. They did not seem to be independent of each other and were not consistent with the class hierarchy we created. For example, the set of event handlers used to simulate the activities of a particular unit to search for targets, select weapons, prepare for a direct fire engagement, and then execute that direct fire engagement differs depending upon whether the unit has a normal radar, special radar, or no radar at all. The existing Janus Simulation System uses the RADAR event handler to carry out the entire procedure if the unit has normal radar. However, it uses the SEARCH, RADAR, and RELOAD event handlers to carry out the procedure if the unit has special radar. Finally the system uses the SEARCH and RELOAD event handlers to conduct the procedure if the unit has no radar at all. We conjecture that this lack of uniformity is due to a series of software modifications made by different people at different times without full knowledge of the software structure. The example also illustrates another problem: the legacy event handlers were not designed to perform independent tasks, and had complicated interactions with each other.

It was necessary to redefine some event categories in order to reduce interdependencies between the event handlers, to factor simulation behavior into more coherent modules, to eliminate redundant coding of the same or similar functions and to take advantage of dynamic dispatching of event handling functions in the object-oriented architecture. Moreover, the Janus system was originally designed to work in isolation, and has since been adapted to interact with other simulation systems. Interactions between the simulation engine and the world modeler (the interface to the distributed simulation network) are performed implicitly within the various event handlers in the existing Janus. Such interactions are made explicit in the new architecture in order to provide a uniform framework to update World Model objects during the simulation. The new architecture uses an explicit priority queue of event objects to schedule the simulation events. We were able to reduce the total number of event handlers needed in the simulation, from 17 to 14, by eliminating identified redundant code (Figure 7).
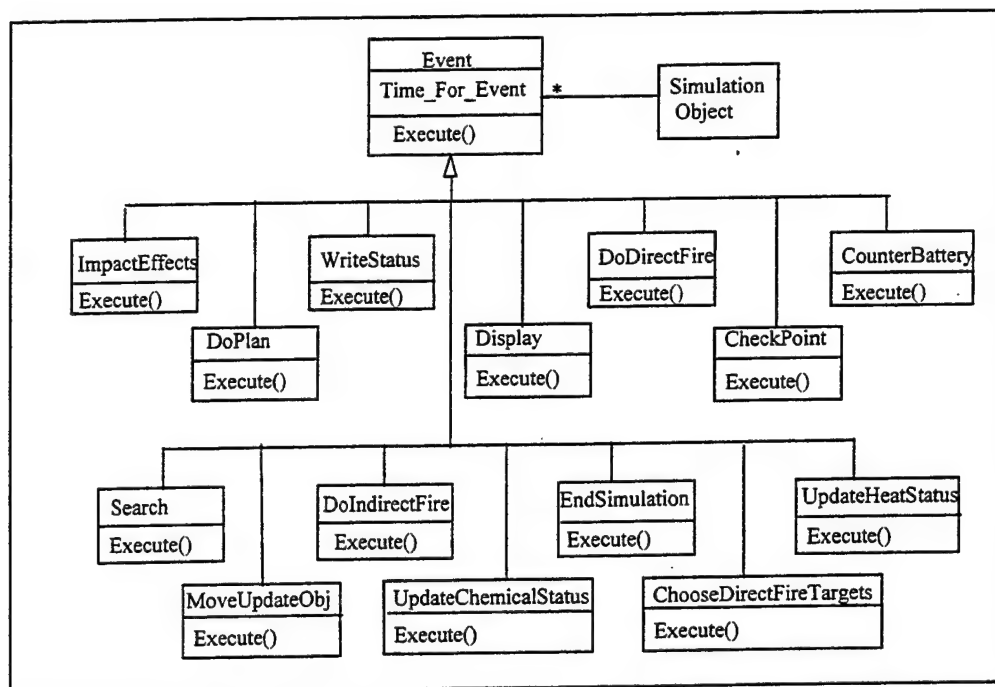
Figure 7. The event class hierarchy.

We tried to make the actions of the new event handlers independent and orthogonal. Independent means that one event handler does not invoke or depend on the action of another. Orthogonal means that the purpose of one event handler is completely separate from that of another. Although our architecture does not completely meet these goals, it comes much closer to them than the legacy design does. We believe that these properties of the architecture are desirable because they impose a partitioned structure on the system that aids future enhancements and modifications. If an enhancement affects only one kind of event, then it becomes relatively easy to isolate the affected part of the code. If suitable naming conventions are followed, relatively low-tech tool support will be adequate for helping system maintainers find the parts of the code that must be understood and modified to make a future change to the system.

Every event has an associated simulation object in the new architecture. This associated object is the target of the event. Depending on the subclass to which an event object belongs, the "execute" method of the event will invoke the corresponding event handler of the associated simulation object. (See [3] for details.) The new event hierarchy enables a very simple realization of the main simulation loop:

```
initialization;
while not_empty(event_queue) loop
    e := remove_event(event_queue);
    e.execute();
end loop;
finalization;
```

Note that this same code is used to handle all of the event handlers, including those for future extensions that have not yet been designed. Event objects with associated simulation objects are created and inserted into the event queue by the initialization procedure, the constructors of simulation objects, and the actions of other event handlers. Depending on the actual event, events are inserted into an event priority queue based on time and priority.

Our newly designed architecture eliminates the need for the simulation loop to know what kind of object it is handling. Thus when adding an object type not yet designed, the simulation loop does not require additional code to invoke the new object's event handlers. By localizing all changes to the newly added object class, our architecture eliminates the possibility of introducing errors into the existing parts of the simulation.

3. Design Validation Via Prototyping

The process of transforming a design developed using the functional approach into an object-oriented design introduces risks of unintentionally altering system behavior. In the context of our case study, the resultant object oriented architecture and the new event dispatching control structure are areas of high risk since they differ significantly from the functional design of the legacy software. UML provides two ways to model behavior. One is to capture the behavior of individual objects over time using state machines, and the other is to capture the interactions of a set of objects in the system using sequence diagrams and collaboration diagrams. While state machines are precise, they only focus on a single object at a time and is hard to understand the behavior of the system as a whole. The sequence diagrams and the collaboration diagrams, on the other hand, lack a formal semantics for precise description of the system behaviors.

One way to reduce the risk is to validate the dynamic behavior of the proposed architecture and to refine the interfaces of subsystems via prototyping at the early design stage. To be effective, prototypes must be constructed and modified rapidly, accurately, and cheaply. Computer aid for constructing and modifying prototypes makes this feasible [15]. The CAPS system is an integrated set of software tools that generate source programs directly from high-level requirement specifications.

Due to time and resource limitations, we developed a prototype for only a very small simulation run, which consists of a single object (a tank) moving on a two-dimensional plane, three event subclasses (*move*, *do_plan*, and *end_simulation*), and one kind of post-processing statistics (fuel consumption).

We developed an executable prototype using CAPS. Figure 8 shows the top-level structure of the prototype, which has four subsystems: *janus*, *gui*, *jaaws* and the *post_processor*. Among these four subsystems, the *janus* and the *gui* subsystems (depicted as double circles) are made up of sub-modules while the *jaaws* and the *post_processor* subsystems (depicted as single circles) are mapped directly to modules in the target language. After entering the prototype design into CAPS, we used the CAPS execution support system to generate the code that interconnects and controls these subsystems. In addition, a simple user interface was developed using CAPS/TAE [21].
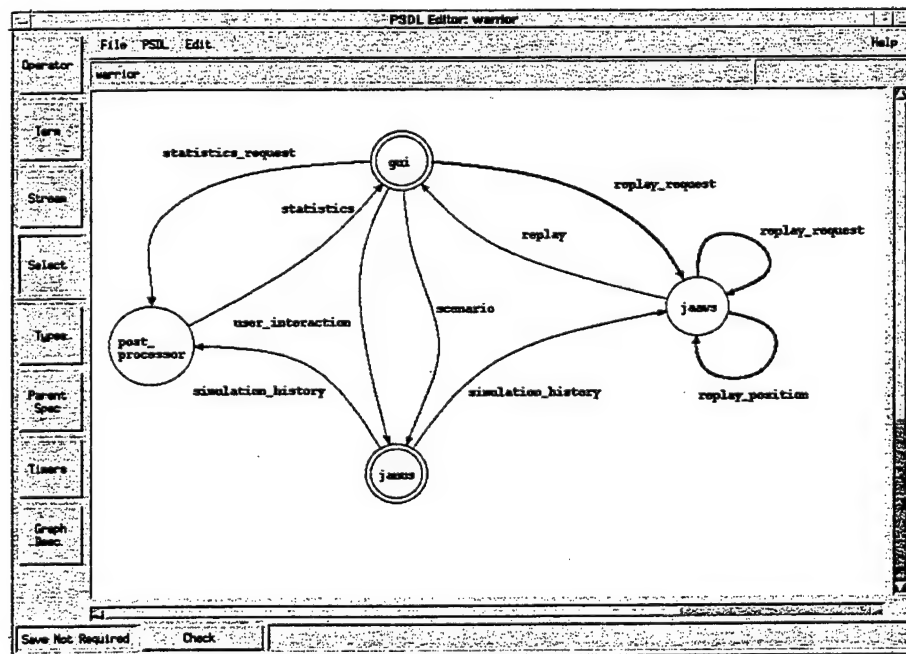
Figure 8. Top-level decomposition of the executable prototype.

The resultant prototype has over 6000 lines of program source code, most of which was automatically generated, and contains enough features to exercise all parts of the architecture. The code that handles the motion of a generic simulation object was very simple, but it was designed so that it would work in both two and three dimensions without modification (currently the initialization and the movement plan of the tank object never call for any vertical motion). The code was also designed to be polymorphic, just as was the main event loop. This means the same code will handle the motion of all kinds of simulation objects

70

without any modifications, including new types of simulation objects that are part of currently unknown future enhancements to Janus and have not yet been designed or implemented.

Our prototyping experiment showed that the proposed object-oriented architecture allows design issues to be localized and provides easy means for future extensions. We started out with a prototype consisting of only two event subclasses (*move* and *end_simulation*) and were able to add a third event subclass (*do_plan*) to the prototype without modifying the event control loop of the Janus combat simulator.

We also demonstrated the use of inheritance and polymorphism to efficiently extend/specialize the behavior of combat units. For example, the *move_update_object* method of a tank subclass uses the general-purpose method from its superclass to compute its distance traveled and a specialized algorithm to compute its fuel consumption. We simply include one statement to invoke the *move_update_object* method of its superclass followed by three lines of code to update its fuel consumption. Moreover, other combat unit subclasses can be added easily to the prototype without the need to modify the event scheduling/dispatching code and usually without modifying existing event handlers.

The issues raised by the design of the prototype also resulted in the following refinements to the proposed architecture:

1. Extend the interface of the *Execute_Event* operation to return the time at which the next event is to be scheduled for the same simulation object, and introduce a special time value "NEVER" to indicate that no next event is needed. The proposed change turns the communication between the event dispatcher and the simulation objects from a peer-to-peer communication into a client-server communication. This change eliminates dependencies of event handlers on event queue details and allows the event dispatcher to use a single statement to schedule all recurring events for all event types.

2. Instead of recording the history of a simulation run in sets of data files, model the simulation history as a sequence of events. The proposed change provides a simple and uniform way to handle history records for all events, and allows the same modular architecture to be used for real-time simulations as well as post-simulation analysis. It also eliminates the need for the write-status event, reducing the number of events still further. This approach provides the greatest possible resolution for the event histories, which implies that any quantity that could have been calculated during the simulation can also be calculated by a post-simulation analysis of the event history, without any loss of accuracy. The only constraint imposed by this design refinement is that the simulation objects in the events must be copied before being included in the simulation history, to protect them from further changes of state as the simulation proceeds. This constraint is easy to meet in a full-scale implementation because the process of writing the contents of an event object to a history file will implicitly make the required copy.

3. It is beneficial to allow null events appear in the event queue. A null event is one that does not affect the state of the simulation, such as a move event for an object that is currently stationary. The prototype version adopted the position that such events should not be put in the event queue, since this corresponds to current scheduling policies in Janus, and appears at first glance to improve efficiency.

Our experience with the development of the prototype suggests that this decision complicates the logic and may not in fact improve efficiency. The current design uses the process *create_new_events* to scan all simulation objects once per simulation cycle to determine if any dormant objects have become active, and if so, schedules events to handle their new activity. The alternative is to have the constructor of each kind of simulation object schedule all of its initial events, and to have each event handler specify the time of next instance of the same event even if there is nothing for it to do currently. Handlers might still set the time of its next event to NEVER in the case of a catastrophic kill; however this is reasonable only if it is impossible to repair or restore the operation of the units that have suffered a catastrophic kill. The reasons why this design change may improve efficiency in addition to simplifying the code are that:

   (a) the check for whether a dormant object has become active is done less often - once per activity of that object, rather than once per simulation cycle,

   (b) executing a null event is very fast - a few instructions at most, so the "unnecessary" null events will not have much impact on execution time, and

   (c) the computation to find and test all simulation objects periodically would be eliminated.

We recommend allowing null events in the event queue, and explicitly scheduling every kind of event for every object unless it is known that there cannot be any non-empty events of that type in any possible future state of the object. For example, under the proposed scheduling policy, immobile or irrecoverably damaged objects would not need to schedule future move events, but those that are

currently at their planned positions would need to do so, because a change of plan could cause them to move again in the future, even though they are not currently moving. The resulting architecture enables a very simple realization of the main simulation.

4. Conclusion

Our conclusion is that substantial and useful computer aid for re-engineering is possible at the current state of the art. Human analysts and domain experts must also play an important part of the process because much of the information needed to do a good job is not present in the software artifacts to be re-engineered. Success depends on cooperation between skilled people and appropriate software tools.

The missing information needed for re-engineering is related to deficiencies of the current system at all levels, from requirements through design and implementation. Thorough and accurate knowledge of these deficiencies is crucial for success. The clients never want the re-engineered system to have the exactly same behavior as the legacy system - if they were satisfied, there would be little motivation to spend time, effort, and resources on a re-engineering project. Even if a system is being re-engineered for the ostensible goal of porting to different hardware, the desired behavior at the interface to the hardware and systems software will be different.

In practical situations, the requirements for the re-engineered system are different from those for the legacy system. Key parts of the requirements for the new system are often missing or incorrect in the legacy documents. Some of that information is present only in the minds of the clients, often fragmented and scattered across members of many different organizations. Communication is a large part of the process, and that communication cannot be automated away, although it can be enhanced by appropriate use of prototyping. We found that the most important communications were those regarding newly recognized requirements issues, and that such recognition were often triggered by discussions between people with different areas of expertise.

Uncertainties about the true requirements play a central role in both re-engineering and the development of new systems. We therefore hypothesized that prototyping could play a valuable role in re-engineering efforts. Our experience in the case study reported here support that hypothesis.

We also found that prototyping can contribute substantially to the process of inventing, correcting, and refining the conceptual structures on which the architecture of the new system will be based. Most legacy systems are too complicated for individuals to understand.

This maze of details hides potential opportunities for simplifying and regularizing the conceptual structure of the system to be re-engineered, and makes it difficult to recognize deficiencies in design and architectural structure. The amplification process implicit in constructing skeletal prototypes helps expose such opportunities.

We found that there are fundamental conceptual errors embodied in the legacy structures and algorithms. Some of those errors were exposed when structural asymmetries and irregularities are discovered in the process of extracting a model of the legacy software. Others were discovered only with the help of the oversimplified models that are common in the early stages of prototyping a proposed new architecture. Constructing a small and simple instance of the proposed architecture raises many of the main design issues, and the simplicity of the model makes it much easier to consider and evaluate alternative designs to find improved structures.

To be effective, prototypes must be constructed and modified rapidly, accurately, and cheaply. The UML interaction diagrams lack the preciseness to support automatic code generation for the executable prototype. This weakness can be remedied by the use of the prototype language PSDL [12, 13] and the CAPS prototyping environment, which provide effective means to model the system's dynamic behavior in a form that can be easily validated by user via prototype demonstration.

References

[1]   A. Aho, "Pattern Matching in Strings", in *Formal Language Theory: Perspectives and Open Problems*, R. Book (editor), Academic Press, NY, 1980, pp. 325-347.

[2]   V. Berzins, M. Shing, Luqi, M. Saluto and J. Williams, *Re-engineering the Janus(A) Combat Simulation System*, Technical Report NPS-CS-99-004, Computer Science Department, Naval Postgraduate School, Monterey, CA, January 1999.

[3]   V. Berzins, M. Shing, Luqi, M. Saluto and J. Williams, "Architectural Re-engineering of Janus using Object Modeling and Rapid Prototyping," *Design Automation for Embedded Systems,* 5(3/4), August 2000, pp.251-263. A preliminary version of the paper also appeared in *Proceedings of the 10th IEEE International Workshop in Rapid Systems Prototyping,* Clearwater Beach, Florida, 16-18 June 1999, pp. 216-221.

[4]   D. Berry, Formal Methods: The Very Idea, "Some Thoughts About Why They Work When They Work," *Proceedings of the 1998 ARO/ONR/NSF/DARPA Monterey Workshop on Engineering Automation for Computer Based Systems,* 1998, pp. 9-18.

[5]   O. Bray and M. Hess, "Reengineering a Configuration-Management System," *IEEE Software,* Vol. 12, No. 1, Jan. 1995, pp. 55-63.

[6]   V. Cabaniss, B. Nguyen and J. Moregenthaler, "Tool Support for Planning the Restructuring of Data Abstractions in Large Systems," *IEEE TSE,* Vol. 24, No. 7, July 1998, pp. 534-558.

[7]   *Janus 3.X/UNIX Software Programmer's Manual,* Prepared for: Headquarters TRADOC Analysis Center, Ft. Leavenworth, Kansas. Prepared by: Titan, Inc. Applications Group, Leavenworth, Kansas, Nov. 1993.

[8]   *Janus 3.X/UNIX Software Design Manual,* Prepared for: Headquarters TRADOC Analysis Center, Ft. Leavenworth, Kansas. Prepared by: Titan, Inc. Applications Group, Leavenworth, Kansas, Nov. 1993.

[9]   *Janus Version 6 User's Manual,* Simulation, Training & Instrumentation Command, Orlando, Florida, 1995.

[10]   *Janus Version 6 Data Base Management Program Manual,* Simulation, Training & Instrumentation Command, Orlando, Florida, 1995.

[11]   S. Jarzabek and P.K. Tan, "Design of a Generic Reverse Engineering Assistant Tool," *Proceedings of the Second Working Conference on Reverse Engineering (WCRE'95),* 1995, pp. 61-70.

[12]   B. Kraemer, Luqi, and V. Berzins, "Compositional Semantics of a Real-Time Prototyping Language," *IEEE Transactions on Software Engineering,* Vol. 19, No. 5, May 1993, pp. 453-477.

[13]   Luqi, V. Berzins, and R. Yeh, "A Prototyping Language for Real-Time Software," *IEEE Transactions on Software Engineering,* Vol. 14, No.10, October 1988, pp. 1409-1423.

[14]   Luqi and M. Ketabchi, "A Computer-Aided Prototyping System," *IEEE Software,* Vol. 5, No. 2, 1988, pp. 66-72.

[15]   Luqi, "System Engineering and Computer-Aided Prototyping," *Journal of Systems Integration - Special Issue on Computer Aided Prototyping,* Vol. 6, No. 1, 1996, pp.15-17.

[16]   Luqi, V. Berzins, M. Shing, M. Saluto, J. Williams, J. Guo and B. Shultes, "The Story of Re-engineering of 350,000 Lines of FORTRAN Code," *Proceedings of the 1998 ARO/ONR/NSF/DARPA Monterey Workshop on Engineering Automation for Computer Based Systems,* Carmel, CA, 23-26 October 1998, pp. 151-160.

[17]   M. Moore and S. Rugaber, "Domain Analysis for Transformational Reuse," *Proceedings of 4th Workshop on Reverse Engineering,* IEEE Computer Society, 1997, pp. 156-163.

[18]   J. Pimper and L. Dobbs, *Janus Algorithm Document, Version 4.0,* Lawrence Livermore National Laboratory, California, 1988.

[19]   L. Rieger and G. Pearman, "Re-engineering Legacy Simulations for HLA-Compliance," *Proceedings of the Interservice/Industry Training, Simulation and Education Conference (I/ITSEC),* Orlando, Florida, December 1999.

[20]   J. Rumbaugh, I. Jacobson and G. Booch, *The Unified Modeling Language Reference Manual,* Addison-Wesley, Reading, MA, 1999.

[21]   *TAE Plus C Programmer's Manual (Version 5.1).* Prepared for: NASA Goddard Space Flight Center, Greenbelt, Maryland. Prepared by: Century Computing, Inc., Laural, Maryland, April 1991.

[22]   J. Williams and M. Saluto, *Re-engineering and Prototyping Legacy Software Systems-Janus Version 6.X,* master's thesis, Naval Postgraduate School, Dept. of Computer Science, Monterey, CA, March 1999.

DCAPS – Architecture for Distributed Computer Aided Prototyping System[1]

Luqi, V. Berzins, J. Ge, M. Shing, M. Auguston[2], B. Bryant[3], B. Kin
Department of Computer Science
Naval Postgraduate School
833 Dyer Road
Monterey, CA 93943 USA
{luqi,berzins,gejun,mantak,auguston,bryant,bkkin}@cs.nps.navy.mil

Abstract

This paper describes the architecture for the distributed CAPS system (DCAPS). The system accomplishes distributed software prototyping with legacy module reuse. Prototype System Description Language (PSDL), the prototyping language, is used to describe real-time software in the DCAPS system. PSDL specifies not only real-time constraints, but also the connection and interaction among software components. Automatic generation of software wrappers and glue is applied for the normalization of data transfer between legacy systems. Implementation of the DCAPS communication layer is based on the JavaSpaces™ library. DCAPS supports collaborative prototype design in a distributed environment.

1 Introduction and objectives

The value of computer-aided prototyping in software development is clearly recognized. It is a very effective way to gain understanding of the requirements, reduce the complexity of the problem and provide an early validation of the system design. Bernstein estimated that for every dollar invested in prototyping, one could expect a $1.40 return within the life cycle of the system development [1]. To be effective, prototypes must be constructed and modified rapidly, accurately, and cheaply. Computer aid for rapidly and inexpensively constructing and modifying prototypes makes it feasible [2].

With advances in wide area networks, there is a need for methods and tools to produce distributed, heterogeneous, and network-based systems that are reliable, flexible and cost effective. Many of these systems are COTS based (commercial off-the-shelf, including "legacy systems"), consisting of a set of subsystems, running on different plat-

forms that work together via multiple communication links and protocols [3][4]. The use of COTS components shifts problems from software development to software integration and interoperability. Builders of COTS-based systems often have no control over the network on which components communicate. They have to work with available infrastructure and need tools and methods to assist them in making correct design decisions to integrate COTS components into a distributed network based system.

Furthermore, as software development has evolved into national and even global cooperative efforts with the explosion of the Internet and World Wide Web, the need for an effective distributed development environment to support such geographically dispersed enterprises became critical. The support is needed both for the distributed design and demonstration of real time system prototypes.

This paper addresses distributed rapid prototyping support for heterogeneous and network-based systems. It presents the underlying architecture to support the specification and automatic generation of codes to integrate and execute COTS components across a heterogeneous network.

2 Motivation and related work

2.1 Prototyping

The demand for large, high quality systems has increased to the point where a quantum change in software technology is needed [5]. Requirements and specification errors are a major cause of faults in complex systems. Rapid prototyping is one of the most promising solutions to this problem. Completely automated generation of prototypes from a very high-level language is feasible and generation of skeleton programming structures is currently

---

common in the computer world. One major advantage of the automatic generation of codes is that it frees the developers from the implementation details by executing specifications via reusable components [5].

An integrated software development environment, named Computer Aided Prototyping System (CAPS) [6] has been developed at the Naval Postgraduate School for rapid prototyping of hard real-time embedded software systems, such as missile guidance systems, space shuttle avionics systems, software controllers for a variety of consumer appliances and military Command, Control, Communication and Intelligence (C3I) systems [7]. Rapidly constructed prototypes are used to help both the developers and their customers visualize the proposed system and assess its properties in an iterative process. The heart of CAPS is the Prototyping System Description Language (PSDL). It serves as an executable prototyping language at the software architecture level and has special features for real-time system design. Building on the success of the Computer Aided Prototyping System (CAPS), the DCAPS model also uses PSDL for specification of distributed systems and automates the generation of interface codes with the objective of making the network transparent from the developer's point of view.

2.2 PSDL and CAPS

PSDL, a prototype description language [8], to describe the real-time software has an open structure so that the user is able to define new properties for software components, such as newly added network configurations. PSDL allows the specification of both input and output guards to provide conditional execution of an operator and conditional output of data. Guards can include conditions on timers that measure duration of system states, and can allow operators to execute only when fresh data has been written to an input stream. Real-time applications, design flexibility, and code reuse motivate the timing and non-procedural control constraints of PSDL. Each time critical operator has a *maximum execution time* constraint, representing the maximum time the operator may need to complete execution after it is fired, given access to all required resources. In addition, each periodic operator has a *period* and a *deadline*. The period is the interval between triggering times for the operator and the deadline is the maximum duration from the triggering of the operator to the completion of its operation. Each sporadic operator has a *maximum response time* and a *minimum calling period*. The minimum calling period is the smallest interval allowed between two successive triggering of a sporadic operator. The maximum response time is the maximum duration allowed from the triggering of the sporadic operator to the completion of its operation. To model distributed systems, PSDL also provides the option of specifying the *maximum delay* associated with any data stream.

CAPS prototypes a software system in the following steps. First, the user selects software components from the reusable component libraries to construct the prototype in a graphic editor. This prototype is saved as a plain text file in PSDL format. The user may also use the graphical user interface (GUI) generator provided by CAPS to create a new GUI for demonstrating and observing the behavior of the prototype. Then, the translator and scheduler work on this PSDL file to generate the wrapper/glue code [9] and dynamic/static schedules [10] respectively. Both the source code of reusable components and automatically generated source code will be compiled together to get the executable. It will be run in the DCAPS environment in order to check both execution correctness and the real-time requirements. As described above, CAPS consists of various prototyping tools to provide all these functionalities. They play different roles during the prototyping process. For example, the scheduler just needs the timing constraints and execution order for every component, while the translator does not care about information other than the network configurations and data type definitions.

In order to automate the integration of COTS in a distributed environment, we need to enhance the modeling capability of PSDL to describe the special operating requirements of the COTS components and the quality-of-service characteristics for the target networks. The enhancement is done via the open syntax provided by the vertex property and edge property of the PSDL graph. Figure 1 shows an example where *the monitor_ environment* and the *temperature_control* operators are realized by COTS components that must run on a Windows NT™ operating system and the *valve_control* operator is realized by a COTS component that must run on a SunOS™ operating system. Furthermore, the *valve_adjustment* data must be transmitted via network links with high security and low latency while the *temperature* data can be transmitted via network links with low security and higher latency. When new properties are introduced in the PSDL descriptions of a prototype, for instance to prototype networked software, some tools must be updated while the rest stay the same. Therefore, the architecture of CAPS must consider the evolution of its own components.

CAPS tools were originally developed in the SunOS operating system for components located on one processor. To avoid the complexity of migrating the whole system to a new operating system, CAPS now has to work in a distributed and heterogeneous environment

2.3 Transaction handling in distributed systems

Building a networked application is entirely different from building a stand-alone system in the sense that many additional issues need to be addressed for smooth functioning of a networked application. Networked systems are

monitor_
environment

MET = 100 ms
PERIOD = 500 ms
PROPERTY os = NT

temperature : celsius
LATENCY = 1000 ms
PROPERTY security = low

fuel : gallons

temperature_
control

MET = 200 ms
MRT = 2000 ms
MCP = 500 ms
TRIGGERED BY ALL temperature
OUTPUT valve_adjustment
    IF |valve_adjustment| > 0.01
PROPERTY os = NT

valve_adjustment : real
LATENCY = 500 ms
PROPERTY security = high

valve_
control

MET = 200 ms
MRT = 2000 ms
MCP = 500 ms
TRIGGERED BY ALL valve_adjustment
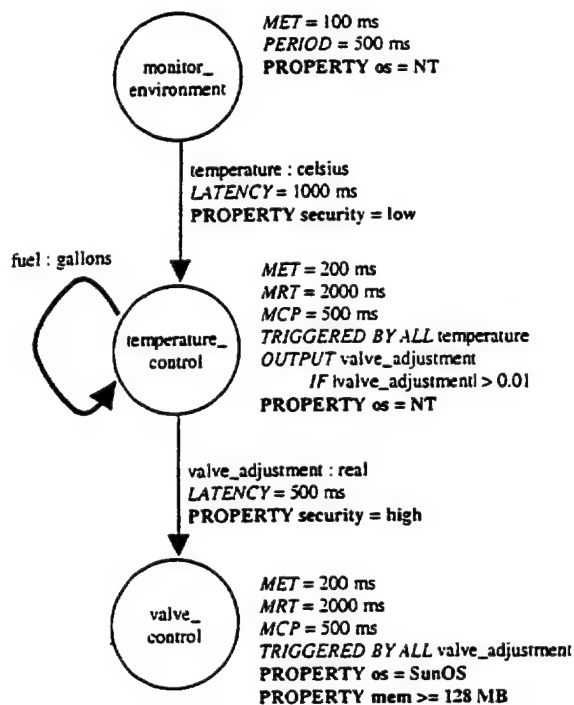PROPERTY os = SunOS
PROPERTY mem >= 128 MB

Figure 1. PSDL specification with additional properties

also susceptible to partial failures of computation, which can leave the system in an inconsistent state.

Proper transaction handling is essential to control and maintain concurrency and consistency within the system. Yang [11] has examined the limitation of hard-wiring concurrency control into either the client or the server. He found that the scalability and flexibility of these configurations is greatly limited. Hence, he presented a middleware approach: an external transaction server, which carries out the concurrency control policies in the process of obtaining the data. Advantages of this approach are: 1) The transaction server can be easily tailored to apply the desired concurrency control policies of specific client applications. 2) The approach does not require any changes to the servers or clients in order to support the standard transaction model. 3) Coordination among the clients that share data but have different concurrency control policies is possible if all of the clients use the same transaction server. PSDL already has a very simple and effective transaction model [12][13] . Transactions are determined by the simple rule that the effect of firing a composite operator must always be equivalent to executing it as a simple atomic action. Optimizations may introduce concurrency and interleave substeps only if that can be done consistently with this rule.

The DCAPS implementation architecture uses the same approach, by using an external transaction manager such as the one provided by SUN in the Jini™ [14] model. All transactions used by the clients and servers are created and overseen by the manager.

2.4 JavaSpaces model

JavaSpaces [14] is a mechanism based upon the Tuple Space model [15] to support coordination among a loosely coupled collection of distributed software systems. Tuples are typed data structures. Collections of tuples exist in a shared repository called a tuple space. Communication takes place in a tuple space shared among several processes; each process can access the tuple space by inserting, reading or withdrawing tuples.

When taking or reading objects, processes use a simple value-matching lookup to find the objects that matter to them. If a matching object is not found immediately, then a process can wait until one arrives. Unlike conventional object stores, processes do not modify objects in the space or invoke their methods directly. To modify an object, a process must explicitly remove it, update it, and reinsert it into the space. During the period of updating, other processes requesting for the object will wait until the process writes the object back to the space. This protocol for modification ensures synchronization, as there can be no way for more than one process to modify an object at the same time. However, it is possible for many processes to read the same object at the same time.

The main benefits of JavaSpaces from the point of view of DCAPS are:

- Spaces are persistent: Spaces provide reliable storage for objects. Once stored in the space, an object will remain there until a process explicitly removes it. This allows a system to perform communication with other systems which may not have begun running yet.

- Spaces are transactionally secure: The JavaSpaces technology provides a transaction model that ensures that an operation on a space is atomic. Transactions are supported for single operations on a single space, as well as multiple operations over one or more spaces.

- Spaces allow exchange of executable content: While in the space, objects are just passive data, however, when we read or take an object from a space, a local copy of the object is created. Like any other local object, we can modify its public fields as well as invoke its methods.

76

- Spaces transcend network topologies: Not only do senders and receivers of messages not need to know each others identities, they also may be located anywhere on the network as long as both have access to the common space.

- Spaces support for time-outs for data.

These properties greatly facilitate the communication layer to be inserted by DCAPS between the various legacy systems being integrated, and ensure the interoperability of these systems.

3 Architecture

3.1 Design time slice of the architecture

The design phase in the DCAPS environment emphasizes the retrieval of PSDL specifications, legacy code (when needed) and distributed resource configuration descriptions both from the server's Project repository and client side directories (Figure 2). DCAPS allow users to model, develop, execute and evaluate prototypes of the proposed systems from different hardware platforms with different operating environments via a web interface shown in Figure 3, where the hyperlinks on the left side of the web-page allow visitors to access information about CAPS, PSDL and request accounts, while the hyperlinks on the right are password protected and can only be accessed by authorized users.
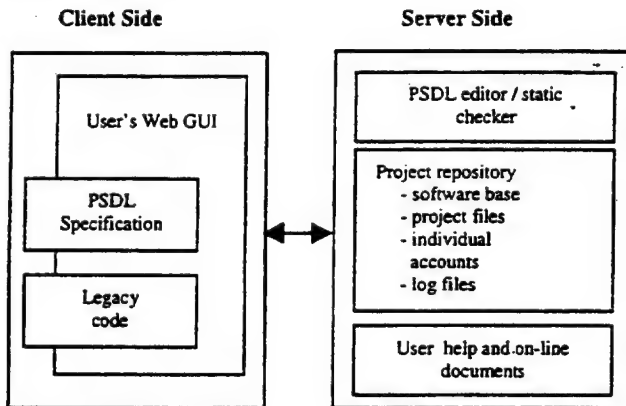
Figure 2. Design Time Slice of Architecture

The Java™-based user GUI ensures that the basic design time tools, such as the graphical PSDL editor, static checker, user help and on-line documentation, and demos are available for clients to run on heterogeneous platforms. An integral part of the Project repository is also individual account information and log files from previous prototyping sessions.

3.2 Compile time slice of the architecture

In the compilation phase of DCAPS, the client-side legacy system and PSDL specification of that system, its interface to the external environment, and the distributed re-
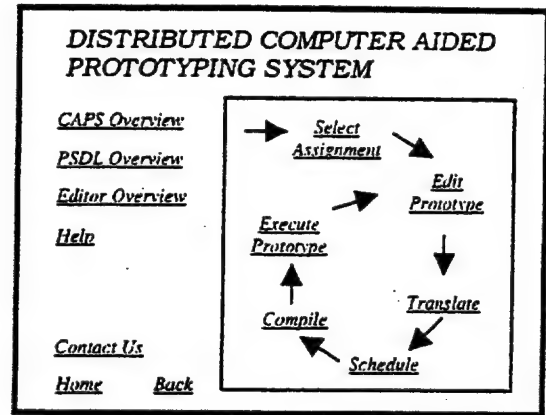
Figure 3. The DCAPS Web Interface

source configuration under which that system is to be run, will be input to the compiling tools residing on the server side (Figure 4). In actuality, these compilation tools may be downloaded to the client side, e.g. using a Java applet, to achieve the compilation.
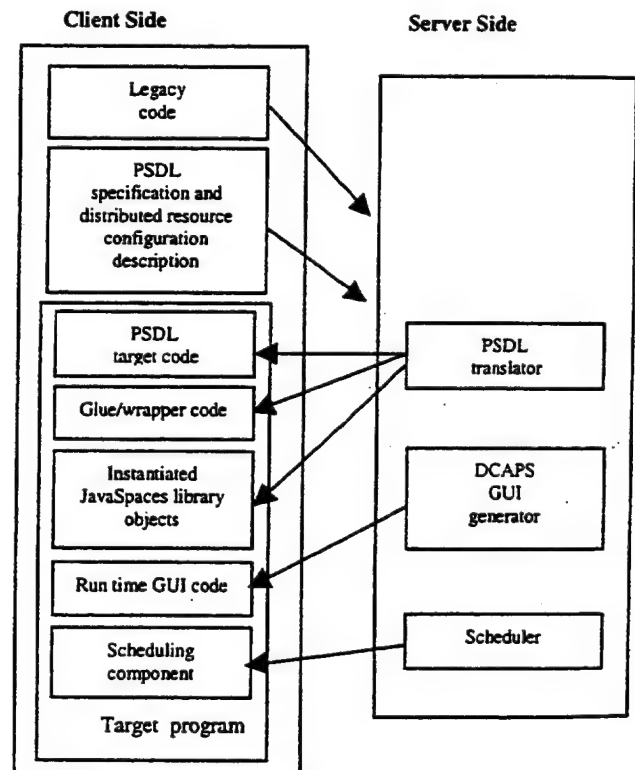
Figure 4. Compile Time Slice of Architecture

Several subsystems that generate source code at various levels are involved in PSDL compilation. The PSDL translator itself produces PSDL target code and wrapper and glue code to connect the PSDL target code to other distributed components. In this process the objects from the JavaSpaces library are instantiated and integrated with the target code. The DCAPS GUI generator produces runtime GUI code which serves as the user interface wrapper

77

for the legacy system. Finally, the static scheduler automatically generates the schedule code component that ensures the target program observes the real-time constraints specified by the PSDL specification.

The existing PSDL data streams are encapsulated as generic Ada™ objects that provide the basic *read* and *write* operations. The actual behavior of the *read* and *write* operations varies depending on whether the data is a FIFO buffer or Sampled buffer. Such encapsulation makes the extension of PSDL data streams to JavaSpaces objects transparent. The only modification is to invoke the JavaSpaces *service registration* operation during the instantiation and initialization of the data objects, and to use the *read*, *write* and *take* JavaSpaces library operations to implement the *read* and *write* PSDL operations.

3.3 Run time slice of the architecture

The current principle of the DCAPS run time architecture is to delegate the inter-process communication layer and scheduling mechanism to the server side (Figure 5). The prototyping session starts at the client side by notifying the server and other clients (by remote login).

The process instances running on one or several client sites use wrappers and instantiated JavaSpaces library objects to send and receive messages. The JavaSpaces library via the underlying tuple space provides the environment for message flow between processes.

The server side also maintains the global logical clock used by the run time scheduler to synchronize process communication and to activate process instances according to PSDL semantics.

Another set of Java-based wrappers for user GUI's generated by DCAPS at compile time provides platform-independent process I/O. Execution traces, i.e. message transaction logs, could be created and stored at the server side for future analysis of the prototyping session.

3.3.1 Synchronization and Logical Clock

The formal real-time model of PSDL is based on the notion of a global clock [12][13]. When operators allocated to different hardware nodes must communicate within strict deadlines, we must account for network delays and imperfect clock synchronization. Our architecture uses local clocks and time reference signals that are broadcast once per iteration of a cyclic schedule to approximate a global clock. Each processor has one such schedule, and all schedules cover the same length of time. The time reference signals determine the local time for the beginning of the schedule at each node. Periodic re-calibration of these time references prevents divergence of the local clocks over long periods of time.
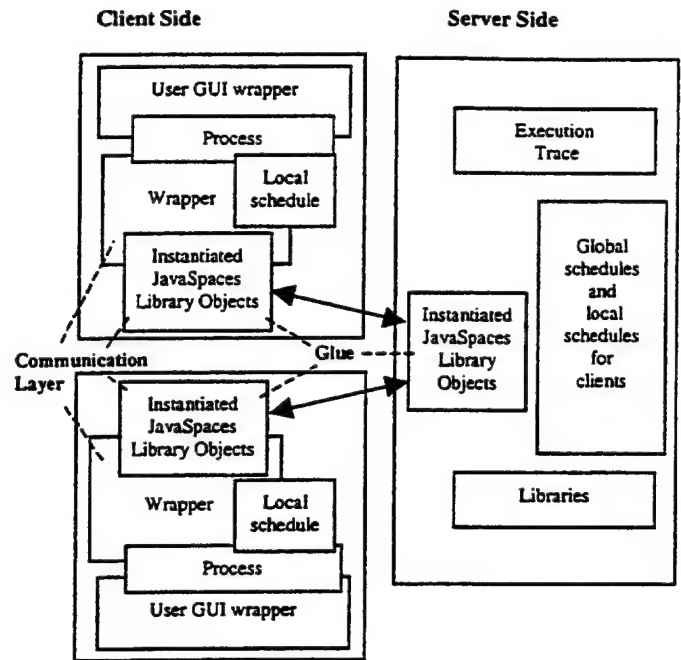
Figure 5. Run Time Slice of Architecture

The worst-case point-to-point network delay bounds initial differences between local clocks. Hardware clocks with stable rates are available and relative drift rates are typically small. The product of the worst-case clock drift rate and the length of the schedule bounds clock drift error. The schedule must account for worst-case clock differences and worst-case clock drift error in addition to worst-case network latency between two nodes when scheduling two operations with a data flow precedence constraint [12][13].

3.3.2 Accurate Simulations on Imperfect Networks

Absolute guarantees of real-time constraints are clearly impossible when designers have no control over the network. In order to simulate a network with guaranteed real time service on an imperfect network, we need the notion of simulated time and supporting mechanisms in the form of:

- Time stamps attached to all communicated data values,

- A time-out period attached to every data communication to work around unbounded delays in the network,

- The mechanism for logical clock synchronization,

- Message buffering for sampled streams based on time-stamp order.

All this results in an accurate approximation of the behavior of a PSDL prototype on a target network with real time service guarantees in a prototyping environment whose networks have no such guarantees.

4 Current state and future work

A Java-based prototype editor has been implemented for the DCAPS. It has been tested in Windows NT, Linux, and Solaris™ environments. Different native interfaces have been implemented as the language wrappers for the Java Spaces-based communication library so that it can be called from applications implemented in different languages. Java Native Interface™ (JNI) makes the library available for C programs, while ActiveX™ wrappers enable Visual Basic™ (VB) programs to call the functions directly. The JNI wrapper makes it possible to create an interface between Ada and C so that programs in Ada can use JavaSpaces services.

The use of centralized control imposes extra communication overhead and creates potential bottleneck on the target heterogeneous system. We plan to conduct empirical studies to analyze the performance of such an approach in support of real-time systems, and investigate ways to relax centralized control by allowing bounded clock drifts among local clocks while still adhering to the constraints imposed by the PSDL timing model.

The current DCAPS scheduler generates a static assignment of the operators of the distributed prototype to the target network. In order to improve the global performance and efficiency of the distributed system, the runtime environment may require a dynamic scheduler to perform runtime load balance and operator reassignment. The mobility provided by the JavaSpaces-based library will support such requirement.

The DCAPS system provides a useful tool for distributed real-time software rapid prototyping in a distributed environment. The wrapper/glue method used in DCAPS can be generalized to system construction and interconnection of legacy systems. By automatically generating the codes for the "wrappers and glue" and providing a powerful environment, DCAPS allows the designers to concentrate on the interoperability problems and issues, freeing them from implementation details. It also enables easy reconfiguration of software and network properties to explore design alternatives. DCAPS is an on-going research project for the development and refinement of its prototyping tools.

References

[1] L. Bernstein, "Forward: Importance of Software Prototyping", *Journal of Systems Integration - Special Issue on Computer Aided Prototyping*, 6(1), pp. 9-14, 1996.

[2] Luqi and W. Royce, "Status report: computer-aided prototyping", *IEEE Software*, 9(6), Nov. 1992, pp. 77 -81.

[3] M. Boasson, *IEEE Software - Special Issue on Architecture*, 12(6), Nov 1995.

[4] S. Mellor and R. Johnson, *IEEE Software - Special Issue on Object Methods, Patterns, and Architectures*, 14(1), Jan/Feb 1997.

[5] Luqi, V. Berzins, "Rapidly prototyping real-time systems", *IEEE Software*, September 1988, pp. 25-36.

[6] Luqi and M. Ketabchi, "A computer-aided prototyping system", *IEEE Software*, 5(2), March 1988, pp. 66 -72.

[7] Luqi, "Computer-aided prototyping for a command-and-control system using CAPS", *IEEE Software*, 9(1), Jan. 1992, pp. 56 -67.

[8] Luqi, V. Berzins, R. Yeh, "A prototyping language for real time software", *IEEE Transactions on Software Engineering*, 14(10), October 1988, pp. 1409-1423.

[9] H. Cheng, *Automated generation of wrappers for interoperability*, Master's Thesis, Naval Postgraduate School, Monterey, Calif., March 2000.

[10] Luqi, M. Shing, "Real-time scheduling for software prototyping", *Journal of Systems Integration - Special Issue on Computer Aided Prototyping*, 6(1), pp. 41-72.

[11] J. Yang, and G. Kaiser, "JPernLite: Extensible Transaction Services for the WWW", *IEEE Transactions on Knowledge and Data Engineering*, 11(4), July/August 1999, pp. 639-657.

[12] Luqi, "Real-Time Constraints in a Rapid Prototyping Language", *Computer Languages*, 18, 1993, pp. 77-103.

[13] B. Krämer, Luqi and V. Berzins, "Compositional semantics of a real-time prototyping language", *IEEE Transaction of Software Engineering*, 19(5), May 1993, pp. 453-477.

[14] E. Freeman, S. Hupfer and K. Arnold, *JavaSpaces: Principles, Patterns, and Practice*, Addison-Wesley, 1999.

[15] D. Gelernter, "Generative Communication in Linda," *ACM Trans. Programming Languages and Systems*, 7(1), Jan. 1985, pp. 80-112.

INTELLIGENT SOFTWARE DECOYS

James Bret Michael and **Richard Riehle**
Naval Postgraduate School, Department of Computer Science
833 Dyer Road, Monterey, CA 93943-5118
bmichael@cs.nps.navy.mil, rdriehle@nps.navy.mil

ABSTRACT
We present an architectural framework for protecting objects from malicious attacks by mobile agents in which the agent tries of circumvent the object-interface to change the behavior of the targeted object. If the agent's interaction with the object-interface contract interface fails preconditions, postconditions, or a class invariant, then the targeted object attempts to both deceive the agent into concluding that the attack has been successful and keep the attacker occupied. The architecture is founded on an abstraction we refer to as an intelligent software decoy: it adapts its behavior to changes in its operating environment. The software decoy is autarkic in that it does not rely on the internal state of other objects to protect itself. The software decoy disguises and defends itself by modifying its object-interface contract at run-time through the use of both polymorphism and late binding. The nature and extent of any change to an object is governed by its class invariant.

KEYWORDS
Agent, broadcast architecture, deception, decoy, distributed system, message architecture, object, security, software, survivability

1. INTRODUCTION

Suppose that there exists a distributed system of thousands of landmines in which each landmine is field-programmable via software hooks. A soldier could broadcast messages to all or a subset of the mines instructing them to either activate or deactivate the electromechanical triggering mechanism, change the compression-pressure threshold value for detonation, or substitute the existing software algorithm for controlling the trigger mechanism with a new algorithm. In addition, the soldier could query the status of the landmines to access the readiness of the minefield to protect against an attack by enemy forces.

On arriving at the software interface of a landmine, a mobile agent would interact with the landmine to reach the goal given to the agent by the owner of the agent. However, if the mobile agent is poorly designed, its flaws may lead the agent to try to interact with the mine-based software in a way that was not intended by the creator of the agent. This could include accidentally tripping a software-based self-destruction mechanism within the landmine. If the software agent is malicious, it may try to sabotage the mine. For example, it might try to alter the mine software so enemy forces can safely cross the minefield. If the software is written in Java, the agent might try to change the behavior of one of the objects or classes. In early versions of the Java Virtual Machine (JVM), such an attack was quite easy to effect due to the fact that a rogue process could insert its own class definition using the same name as the original predefined Java class [12].

The minefield example illustrates the potential for mobile agents to modify the software-controlled behavior of a distributed system or a subset of the components of the system in a mischievous way. One approach to protecting a system such as the distributed system of landmines is to both encrypt the messages and to authenticate the mobile agents to the objects. However, McHugh and Michael have identified some of the challenges in managing cryptographic keys in such systems, especially when group membership (e.g., subgroups of the mines) changes frequently [13]. Moreover, an authenticated mobile agent may have been compromised, or its creator, who at one time was trustworthy, may no longer be so. In summary, encryption and authentication do not address the issue of discovering and responding to the goals or actions of mobile agents.

Another approach to protecting the distributed system from mobile agents would be to require that the agents only interact with the landmines via a well-defined object-interface contract. This is the well-known design-by-contract model described in the work of Meyer [15]. However, a malicious agent would likely try to bypass the contract to modify the behavior of the targeted object. Thus, precondition assertions for controlling access to the object may only be effective at thwarting the actions of non-malicious agents, that is, agents whose flawed design induces unintended interactions with objects through the interfaces to these objects. This is known in the epigrammatic world as "Locks are intended to keep honest people honest."

In this paper we present an architectural framework for use in protecting objects from malicious attacks by mobile agents, in particular, attacks in which the attacker tries of circumvent the object-interface in order to change the behavior of the targeted object. The architecture is founded on an abstraction we call an intelligent software decoy. The software decoy is intelligent in the sense that it adapts its behavior to changes in its operating environment. The software decoy is autarkic in that it does not rely on the internal state of other objects to protect itself. The software decoy disguises and defends itself by

altering its contract at run-time through the use of polymorphism. The nature and extent of any change to an object is governed by its class invariant. The invariant, and in some cases, a postcondition, will ensure that no mischief has occurred during the execution of the object's run-time code.

2. SOFTWARE DECOYS

A decoy is intended to deceive something or someone into believing it is the object it advertises itself to be. Therefore, the creator of a decoy must actualize the decoy as much as possible to complete the deception. The more the external observer is deceived, the better the decoy is performing its role. Daniel and Herbig define deception as the "deliberate misrepresentation of reality done to gain a competitive advantage" [4].

When a duck hunter deploys decoys on a lake, those decoys are painted to resemble the species of duck being pursued. If the decoys can be made to move about, the deception may be more effective: the real ducks will think that the decoys are also real since the decoys appear to be paddling through the water. In this case, the effectiveness of the decoys need only be good enough so as to draw the real ducks within shotgun range.

A software decoy has some of the same properties as the physical decoy. It certainly has the same objective: deception. If the decoy is intelligent, it can continually deceive the target of the deception into action that accomplishes several goals. In the case of an attack or the deployment of countermeasures executed by an attacker, one of the goals of the owner of the decoy is to protect the actual entity being shielded from attack and anti-decoy countermeasures.

Another goal, in the context of an attack, is to ensure that every attack reveals the presence of an attacker. In this way, the decoy can use its own intelligence to deploy more decoys and to alert the actual entities that an attack has been attempted. As more decoys are deployed, their creator can also alter their own characteristics so that the decoys appear to be different from the one originally attacked.

In an ideal situation, the decoys will be able to adopt a chameleon-like character that allows them to appear to be different as other decoys and attackers change form. In the context of software decoys, this model of decoys raises the concept of intelligent agents to a new level of sophistication. It requires that both the interfaces and the objects be polymorphic, that is, the contract for each object must be polymorphic. Consequently, any message to a decoy can be encrypted, but the decoy will have its own knowledge of the encryption scheme based on the parameters of the polymorphic message. Successful execution of the decoy will require satisfying the precondition, the invariant, and the postcondition. Since the postcondition is internal to the object, it is not easily compromised even with dynamic patching schemes.

3. PRIOR RESEARCH

The general notion of a software decoy is not new. For example, the term "decoy" has been used in the context of reasoning with incomplete information in multiagent systems. According to Zlotkin and Rosenshein [25],

> One obvious way in which uncertainty can be exploited can be in misrepresenting an agent's true goal. In a task oriented [sic] domain, such misrepresentation might involve hiding tasks, or creating false tasks (phantoms, or decoys), all with the intent of improving one's negotiating position. The process of reaching an agreement generally depends on agents declaring their individual task sets, and then negotiating over the global set of declared tasks. By declaring one's task set falsely, one can in principle (under certain circumstances), change the negotiation outcome to one's benefit.

This earlier research indirectly addresses the Byzantine Generals problem [11] in that they try to construct incentive-compatible negotiation mechanisms such that "no agent designer [sic] will have any reason to do anything but make his agent declare his true goal in a negotiation." In contrast to the work of Zlotkin and Rosenshein, in which interaction between agents was investigated, we explore the use of software decoys in the context of the interaction between agents and software components.

Turing introduced the "imitation game" [23], now known as the Turing test, for testing the intelligent behavior of software. The participants in the test consist of a computer, a human, and an interrogator. The goal of the interrogator, who is a human subject, is to distinguish between the computer and the human with whom he or she carries on a conversation. The identity of the respondent, that is the computer or human, is hidden from the interrogator. The measure of intelligent behavior of the software system is the percentage of time that the interrogator cannot correctly distinguish between the response of the computer, which simulates a human response, and that of the person typing responses. Thus, the game is one of deception: programming a machine to deceive, via impersonation, a human into believing that the machine he or she is conversing with is also a human being.

In contrast to the approach taken by Turing to test for intelligent behavior reasoning by a computer, Goldberg [7] attempts to address questions of intelligent reasoning by computers, arguing that a computer cannot deceive itself. His argument relies on a "commonsense view of the mind," that is, that a computer cannot possess beliefs or self-knowledge, as can a human.

However, Goldberg does not address the issue of whether one computer can deceive another. In our own work, we argue that it is possible for a software component to deceive an agent by creating a deception based on either direct inspection of the internal state of the other agent, or alternatively, assessing the intentions of the agent by monitoring the agent's behavior. In addition, we subscribe to the theory posed by Hirstein that self-deception can be due to conflicts other than between beliefs, namely, a "conflict between two representations, a 'conceptual one' and an 'analog' one" [8]. Our conception of a decoy is one in which a decoy, agent, or other type of software can itself possess conflicting representations.

In [5], examples are presented of the use of deception in military campaigns dating back thousands of years. In [3], Cohen presents a classification of defenses for information systems, in which one of those defenses is deception:

> *Defence 98: deceptions.* Typical deceptions include concealment, camouflage, false and planted information, reuses [*sic*], displays, demonstrations, feints, lies, and insight (Dunnigan, 1995). Examples include facades used to misdirect attackers as to the content of a system, false claims that a facility or system is watched by law enforcement authorities, and Trojan horses planted in software that is downloaded from a site. Deceptions are one of the most interesting areas of information protection, but little has been done on the specifics of the complexity of carrying out deceptions. Some work has been done on detecting imperfect deceptions.

Cohen has explored this class of defense for use in protecting computing resources in a distributed system. He refers to such protection techniques as "defensive network deceptions" [1], and has attempted to develop formal models of defensive deceptions and the types of attackers for which these deceptions are to be used. In one of these models, the attacker is characterized as an agent "who believes that information systems are vulnerable and [the attacker] has finite resources to attack" the systems. In this model, the attacker relies on intelligence reports about the information systems in order to identify and choose a specific vulnerability of the system to target, and that the attacker will not attack unless it believes that "there exists an exploitable weakness of value." In the other model Cohen presents, the attacker and defender are both assumed to believe that all systems of positive non-zero economic worth have at least one exploitable weakness.

Cohen introduces six goals for defensive network deceptions [1]; they are to make the following:

1. Likelihood of any individual intelligence probe encountering a real vulnerability low.
2. Likelihood of any individual intelligence probe encountering a deception high.
3. Time to defeat a deception infinite.
4. Time to detect a vulnerability once a deception is encountered from a given attack location infinite.
5. Time to detect an intelligence probe against a deception very small.
6. Time to react to an intelligence probe against a deception very small.

These goals, to some extent, have been incorporated into the Deception Toolkit (DTK) [2]. Prior to the emergence of the DTK, the most widely used type of tool for defensive network deception was the honey pot, which is still used today. A honey pot is a decoy that is placed in a highly visible location within an information system so as to draw the attention of attackers. According to Cohen, honey pots have not proved to be very effective at influencing the decision making of an attacker because each honey pot "consumes such a small portion of the overall intelligence space and has little effect on altering the characteristics of the typical intelligence probe" [1].

The DTK distributes deceptions throughout the network to be protected, with the deceptions utilizing unused network-system resources. An example of a deception that can be created using the DTK is to populate the network with IP addresses masquerading as addresses of valuable system resources: the fake IP addresses and dummy resources associated with them serve as decoys. The DTK has evolved from a simple extension to honey-pot systems to incorporate techniques to both increase the size of the search space (i.e., for a real versus decoy service) and the sparseness of actual vulnerabilities. Cohen has also used the DTK as an experimental apparatus for testing strategies to improve the quality of deceptions. The strategies he lists in [1] include the following: injecting synthetic network traffic into the network, reconfiguring the deception network over time, injecting synthetic information about the organization and its constituents into the system, and using real systems rather than software sandboxes as decoys.

Moose [16], like Cohen, has tried to model deception from a systems view. He explicitly models the evolution of pairs of stimuli and responses between the defenders of a system who are using deception techniques and that of the attackers. The modeling paradigm is intended to capture deception and counter-deception scenarios, the plans of actors (i.e., defender and attacker), uncertainty associated with intelligence information, feedback loops, and the risk models of actors.

The Denial and deception Analyst Workbench (DAWS) [10] is an interactive system used by intelligence analysts to maintain denials and deceptions, in other words, cover stories. DAWS consists of a set of integrated tools, managed by an expert system. DAWS pre-processes raw intelligence data so that it can be automatically forwarded to analysts based on pattern matches on their information-needs profiles. The other tools help the user manage denials and deceptions that are perpetuated for a target audience. DAWS and DTK are similar in that they both are designed with the human in the loop.

The development of counter-deception techniques has been a very active area of research in the information theory community. For example, in the 1970s, Gilbert et al. [6] explored the use of codes to detect evidence of deception on the part of an opponent that tries to intercept or change messages between a transmitter and its intended receiver. The opponent tries to capture message streams on a channel without letting the original transmitter or the intended receiver know that the message has been captured. The typical attack scenario involves a rogue process, such as a Trojan horse, that redirects message traffic on trusted channels or via a covert channel (i.e., a channel that bypasses the information system's reference monitor). The opponent may raise the deception to an even higher level of sophistication by implementing a man-in-the-middle attack. In such an attack, the opponent captures a message, m, modifies the captured message, yielding m', and makes m' looks as though it has not been tampered with. The opponent impersonates the original transmitter while forwarding m' to the receiver that the original transmitter had intended m to reach.

Recent advances in information theory, such as those reported in [9, 14, 20] have produced authentication-coding schemes for detecting deception in authentication channels with single or multiple usage (i.e., without changing the key after each message is sent). The authentication codes are used to derive the lower bounds on the probability that an opponent will successfully deceive the receiver via substitution or impersonation.

Tognazzini [22] has investigated constructive uses of deception for designing human-computer interfaces. He compares the art of illusion, as practiced by magicians, to the illusions created by the designers of graphical user interfaces, that is, the virtual reality that the user of the interface perceives. Some of the techniques that he explores are misdirection, attention to detail, and the manipulation of time. He concludes his essay with a discourse on the concept of a threshold of believability (on the part of the user of a graphical user interface) and the ethics of impersonation, in the form of anthropomorphism (i.e., software agents impersonating humans).

4. INTELLIGENT SOFTWARE-DECOY ARCHITECTURE

In this section we characterize the components and connections of our software-decoy architecture.

Components, Named Interfaces, and Reuse

We treat intelligent software decoys as objects within components, following the usage by Szyperski of the terms "component," "object," and "interface" to describe component-based software architectures [21]. The connectors between components are named interfaces. There is no requirement for the name that a decoy advertises to other components to be unique. The interface of a decoy consists of an ordered list of arguments. The arguments can be either primitive types or object classes. In the latter case, the argument supports polymorphic types. Each class is composed of its own arguments and behavior. The arguments are used to access the methods of objects within a component.

A software decoy can replicate itself, using the same name for the cloned components. Mobile agents cannot distinguish a component from its decoy. In order for components to be able to distinguish amongst themselves, one could implement the architecture using a single address space operating system such as Sombrero [19], or possibly a distributed operating system that supports object-request brokers, such as StratOSphere [24].

Dynamic Component Interface

An intelligent software decoy can change the form of its contract interface at run-time. The modification of the form of a decoy's interface is supported by polymorphism; that is, the component inherits its interface from its parent class. The modification of the interface can involve changing one or more of the following: the number of arguments, the order of arguments, or the data type or class of arguments. The number of possible combinations of input arguments, in theory, is infinite, as is the number of class derivations. The permutation of arguments to introduce randomness into a system is not new. For example, Rothstein introduced the idea of using permuted arguments as a form of decoy in his work on message opacifiers [18].

In addition to permuting the ordering of the arguments and changing the quantity and type of arguments, randomness is injected into the interface by padding the input-argument list with one or more dummy arguments. While the total number of arguments is held constant, the position of the dummy arguments in the argument list can be changed, as can the data types of any of the arguments. The number of permutations, denoted by P, of the input-argument list for this strategy is

$$P_{m,n} = k^m \cdot (m+n)!$$

(Eq. 1)

where m is the total number of dummy arguments, n is the total number of legitimate arguments, and k is the number of unique data types (both primitive and class-based) from which to assign a type to a dummy argument.

A mobile agent computes an argument list for an object it wants to access and passes that list along with authentication information to the interface of the target object. After the agent is authenticated to the object, the object verifies that the argument list that the agent passed to it is correct.

Definition (Correct agent-generated argument list): An agent-generated argument list is correct if and only if the number, ordering, and type of these arguments exactly match those of the target object's interface.

If the agent-generated argument list is correct, then the client where the object resides checks the access control list to determine whether the agent holds the permissions to access the method (e.g., execute the method locally or export the method for remote execution).

Protection of Object Behavior from Unauthorized Modification

Preconditions, postconditions, and class invariants govern the behavior of an intelligent software decoy. If the pre- or postconditions fail during an interaction with a mobile agent, then the decoy either aborts the requested call or both raises an exception and unwinds to the caller. An alternative policy to raising exceptions is to retry the operation with a new set of data. The class invariants protect decoys from having their behavior modified in an unauthorized way. An agent cannot modify the behavior beyond the extent to which such modification is permitted by the parent class of the decoy.

Randomness can also be introduced into the design of the decoys by allowing the preconditions on the invocation of methods of a component to vary.

$$P_{m,n,q} = k^m \cdot (m+n+q)! \qquad \text{(Eq. 2)}$$

where q is the number of unique preconditions in the sample space. We do not allow for the class invariant to be permuted.

Polymorphic Types

As mentioned earlier, component interaction is based on a contract that is controlled by assertions (i.e., preconditions) as well as by a polymorphic type. The polymorphic type permits a late-binding of the message interaction. The preconditions require certain characteristics to be satisfied for each interaction to be carried forward. Preconditions are not a strong enough mechanism for all circumstances. They are particularly ineffective at guarding against mischievous action.

Polymorphic types are a little more interesting. We declare that certain parameters can have different characteristics within some accepted range of types. The types themselves may carry a set of encryption features as well as other encoding that makes them less likely to be compromised by an attacker.

An important difference in a software decoy is when the encryption error is rejected. Ordinarily, if a password fails on a routine, that routine rejects the attempt at entry. The software decoy instead, lets mischief proceed unnoticed by the attacker. Instead of repelling the attack, the software decoy engages it without revealing that its action is benign. This could be called the Venus flytrap model. This pleasant looking little flower lets its prey enter, enjoy the fragrance of its pollen, and encloses it for a tasty meal.

If the precondition is satisfied and the mischief is in the form of a patch, both the software decoy and the non-decoy are defended by the invariant and the postcondition. Once again, if the invariant fails within the decoy, the attacker is never notified. If the postcondition fails, we apply a kind of software *jiu jitsu* within that decoy. This means we allow the attacker to believe it has been successful in overpowering the defenses while tumbling it harmlessly through the code instead of letting it forward any messages to other agents. Our approach to deception is a cross between ambiguity-increasing (A-type) deception [4], in which the decoy seeks to ensure that the "level of ambiguity always remains high enough to protect the secret of the actual operation," and misleading (M-type) deception [4], which entails reducing ambiguity by "building up the attractiveness" of a decoy, thus causing the attacker to concentrate its resources on the decoy.

Exchange of Roles

A legitimate object can exchange its role with one of its decoys. It initiates an exchange of roles when it detects an anomalous behavior of a mobile agent.

Definition (Anomalous behavior of a mobile agent): An anomalous behavior of a mobile agent is one in which a request for access to a legitimate object by a mobile agent fails the test of authentication, test for correctness of the agent-generated argument list, or the check for the necessary access permissions.

Policy 1 (Exchange of roles): If a legitimate object detects anomalous behavior of a mobile agent, then the legitimate object exchanges roles with one of its decoys.

The purpose of Policy 1 is to free up the legitimate object from processing legitimate requests so that it can take on the role of a software decoy, in particular, gather information about the mobile agent.

Observation-Inference Component

The software decoy tries to determine the nature of a mobile agent's interaction with it in order to respond appropriately to the mobile agent. The software decoy records the messages passed to its interface by the mobile agent. The software decoy has a pattern recognition capability for distinguishing between whether an anomalous behavior exhibited by a mobile agent is due to an error in the mobile code or an attack by that agent.

Response Component

The role of design-by-contract [15] is critical. There can be a failure of the precondition, in which case, we must have a response policy for precondition violations. In general, failure of a precondition means the agent will not do any of its work. The policy question remains for each agent: what action is appropriate when the precondition fails? A bad precondition may originate from a benign source or may represent an attempted attack. At the very least, we keep track of such failures. Failure of the invariant or postcondition intuitively represents a higher probability of an attack on the agent. In particular, the failure of the postcondition should trigger the self-modifying behavior of the decoy.

The policy for responding to a mobile agent is embedded in the software decoy. The person or organization that owns the software decoy might specify the following policies:

Policy 2 (Containment by decoy): If a mobile agent, due to a software error in its code, passes an incorrect argument list to the to the software decoy or the real object, then the decoy should activate its containment countermeasure, rather than actively attack the mobile agent.

Policy 3 (Counterattack by decoy): If the mobile agent intended to attack the object, then the object should not under respond by treating the interaction as being due to a software error.

In this scenario, an active attack on a non-malicious mobile agent could trigger a counterattack by the mobile agent or the mobile agent's coordinating agent. Rather, in this scenario, the policy embedded in the object might be to apply containment countermeasures that involve an active attack on the mobile agent, the applet that generated the agent, or even the user who invoked the applet.

5. LANGUAGE SUPPORT FOR INTELLIGENT SOFTWARE DECOYS

We believe that Eiffel is a natural choice of programming languages for implementing intelligent software decoys, at least for the purposes of initial experimentation with such decoys. In contrast to Ada, for example, Eiffel provides explicit support for design-by-contract in the form of built-in language constructs for specifying preconditions and postcondtions in routines. Returning to the example of software-controlled landmines, the software routines for enabling the triggering mechanism of a landmine could be protected by wrapping the routines with preconditions and postconditions as follows:

```
class LANDMINE
-- A landmine with identification number and a status (armed or disarmed)
feature
  definitions
arm_mine(parameter list) is
-- routine for arming the trigger mechanism of a mine
require
  preconditions
do
  operations
ensure
  postconditions
invariant
  invariants
end
```

Moreover, Eiffel provides for inheritance of the assertions from ancestor classes by a descendant class, which is need to preserve the integrity of the software contracts for the software decoys that are generated by a software component. However, not all Eiffel systems support the full range of the levels of run-time assertion monitoring.

6. CONCLUSION

Our approach to deception is different from that proposed in [1] in that we introduce the use of software contracts and polymorphism to create and manage software decoys. The software contracts are used to specify security policy and mediate the interaction under policy between the intelligent software decoy and attacker: the postconditions and invariants place fail-safe constraints on the behavior of the decoy, thus permitting the decoy to allow the attacking mobile agent to interact with the decoy while containing the agent. The class invariant makes it impossible for the attacker to modify the behavior of a decoy, while polymorphism permits the decoy to change its appearance, in the form of preconditions, to the attacker. Moreover, the intelligent software decoys populate the entire system space; that is, every software component can switch roles at run-time—from non-decoy to a decoy, and vice versa—and replicate itself. In addition, the decoys can operate in an autonomous manner, due to their autarkic nature, or they can communicate their intentions to other software components to coordinate their actions to either deceive attackers or trace the source and nature of the attack.

7. FUTURE WORK

We are in the process of refining the mathematical formulation of the software-decoy architecture, in addition to typing decoys, such as distinguishing between "volunteer" and "drafted" decoys. We have also begun investigating the technical feasibility of realizing the various concepts we have introduced: we are implementing intelligent software decoys using Eiffel and intend to perform analyses of the behavior of decoys under various scenarios.

In addition, we are exploring ways to apply intelligent software decoys in distributed databases in which lightweight objects perform queries on multidatabases. In particular, we are developing an example of how intelligent software decoys can be used in the DBMS-Aglet Framework proposed by Papastavrou, Samaras, and Pitoura [17].

ACKNOWLEDGEMENTS

REFERENCES

1. Cohen, F. A mathematical model of simple defensive network deceptions. *Computers & Security* 19, 6 (2000), 520-528.
2. Cohen, F. A note on the role of deception in information protection. *Computers & Security* 17, 6 (1998), 483-506.
3. Cohen, F. Information system defences: a preliminary classification scheme. *Computers & Security* 16, 2 (1997), 94-114.
4. Daniel, D. C. and Herbig, K. L. Propositions on military deception. In Kaniel, D. C. and Herbig, K. L., eds., *Strategic Military Deception*. Pergamon Press, New York, 1982, pp. 3-30.
5. Dunnigan, J. F. and Nofi, A. A. *Victory and Deceit*. William Morrow & Co., New York, fifth ed., 1995.
6. Gilbert, E. N., MacWilliams, F. J., and Sloane, N. J. A. Codes which detect deceptions. *Bell Syst. Tech. Jour.* 53, 3 (1974), 405-424.
7. Goldberg, S. C. The very idea of computer self-knowledge and self-deception. *Minds and Machines* 7, 4 (Nov. 1997), 515-529.
8. Hirstein, W. Self-deception and confabulation. *Jour. Philosophy of Science* 67, 3 (Suppl. S, Sept. 2000), S418-S429.
9. Johansson, T. Lower bounds on the probability of deception in authentication with arbitration. *IEEE Trans. Inf. Theory* 40, 5 (Sept. 1994), 1573-1585.
10. Kisiel, K. W., Rosenberg, B. F., and Townsend, R. E. DAWS: Denial and deception analyst workstation. In *Proc. Internat. Conf. on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems*, Vol. II, IEEE (Tullahoma, Tenn., June 1989), 640-644.
11. Lamport, L, Shostak, R, and Pease, M. Byzantine Generals problem. *ACM Trans. Programming Languages and Systems* 4, 3 (1982), 382-401.
12. McGraw, G. and Felton, E. *Java Security: Hostile Applets, Holes, and Antidotes*. John Wiley & Sons, New York, 1996.
13. McHugh, J. and Michael, J. B. Secure group management in large distributed systems: What is a group and what does it do? In *Proc.New Security Paradigms Workshop*, ACM (Caledon Hills, Ont., Sept. 1999), 80-85.
14. Meijer, A. R. Deception in authentication channels with multiple usage. In *Proc. IEEE South African Symp. on Communications and Signal Processing*, IEEE (Matieland, South Africa, Oct. 1994), 60-67.

15. Meyer, B. *Object-Oriented Software Construction*. Prentice-Hall, Upper Saddle River, N.J., 1998.

16. Moose, P. H. A systems view of deception. In Kaniel, D. C. and Herbig, K. L., eds., *Strategic Military Deception*. Pergamon Press, New York, 1982, pp. 136-150.

17. Papastavrou, S., Samaras, G., and Pitoura, E. Mobile agents for world wide web distributed database access. *IEEE Trans. Knowledge and Data Engin.* 12, 5 (Sept.-Oct. 2000), 802-820.

18. Rothstein, J. Parallel processable cryptographic methods with unbounded practical security. In *Proc. Internat. Symp. on Inf. Theory*, IEEE (Ithaca, N.Y., Oct. 1977), 43.

19. Skousen, A. and Miller, D. The Sombrero single address space operating system prototype: A testbed for evaluating distributed persistent system concepts and implementation. In *Proc. Internat. Conf. on Parallel and Distributed Processing Techniques and Applications*, CSREA Press, (Las Vegas, Nevada, June 2000), 557-563.

20. Smeets, B. Bounds on the probability of deception in multiple authentication. *IEEE Trans. Inf. Theory* 40, 5 (Sept. 1994), 1586-1591.

21. Szyperski, C. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1999.

22. Tognazzini, B. Principles, techniques, and ethics of stage magic and their application to human interface design. In *Proc. Conf. on Human Factors in Computing Systems*, ACM (Amsterdam, Neth., Apr. 1993), 355-362.

23. Turing, A. M. Computing machinery and intelligence. *Mind* 59, 236 (Oct. 1950), 433-460.

24. Wu, D., Agrawal, D., and El Abbadi, A. StratOSphere: mobile processing of distributed objects in Java. In *Proc. Fourth Annual Internat. Conf. on Mobile Computing and Networking*, ACM (Dallas, Tex., Oct. 1998), 121-132.

25. Zlotkin, G. and Rosenschein, J. S. Mechanism design for automated negotiation, and its application to task oriented domains. *Jour. Artificial Intelligence* 86, 2 (Oct. 1996), 195-244.

Enhancements & Extensions of
Formal Models for Risk Assessment in Software Projects

Michael R. Murrah
mrmurrah@nps.navv.mil

Luqi
Luqi@cs.nps.navv.mil

Craig S. Johnson
csjohnson@dcmdw.dcma.mil

Naval Postgraduate School
2, University Circle
Monterey, CA 93943 USA

Abstract

Over the past 40 years limited progress has been made to help practitioners estimate the risk and the required effort necessary to deliver software solutions. Recent developments improve this outlook, one in particular, the research conducted by Juan Carlos Nogueira [1]. Dr. Nogueira developed a formal model for risk assessment that can be used to estimate a software project's risk when examined against a desired development time-line. This model is based on easily obtainable software metrics. These metrics are quantifiable early in the software development process.

Dr. Nogueira developed his model based on data collected from a series of experiments conducted on the Vite'Project simulation [2]. This unique approach provides a starting point towards a proven formal model for risk assessment, one that can be applied early in the software development lifecycle. Approaching software risk estimation has never previously been successfully accomplished in this manner.

The proposed research will provide definitive evidence that software risk assessment can be conducted early in software development using quantifiable metrics and simple techniques. Enhancements will be made to Dr. Nogueira's model, based on calibrations against post-mortem projects. These enhancements will result from many threads of research; extension of input metrics, increased number of simulation runs, simulation scenarios based on actual projects, and the introduction of a "gearing factor". Ultimately, the research will yield an improved risk assessment model, one that has been validated against thousands of post-mortem projects, having applicability to any software development activity.

1. Introduction

The current state of the art techniques of risk assessment rely on checklists and human expertise. This constitutes a weak approach because different people could arrive at different conclusions from the same scenario. The difficulty of estimating the duration of projects applying evolutionary software processes adds intricacy to the risk assessment problem.

2. Dr. Nogueira's Risk Assessment Model

Dr. Nogueira's research introduces a formal method to assess the risk and the duration of software projects automatically, based on measurements that can be obtained early in the development process. The method has been designed according to the characteristics of evolutionary software processes, and utilizes quantifiable indicators such as efficiency, requirement volatility and complexity. The formal model, based on these three indicators estimates the duration and risk of evolutionary software processes. The approach introduces benefits in two fields:

a) Automation of risk assessment.
b) Early estimation methods for evolutionary software processes.

Dr. Nogueira developed four software risk estimation models that show great promise in determining a software projects' associated risk early in the software development life cycle. The models accomplish early estimation by utilizing a set of quantifiable metrics that can be collected from the beginning of project development. In actuality, the requirements volatility metric is an estimation during the first development cycle and during subsequent development cycles is quantifiable. After each iteration of software development, the required

input metrics can be applied to the model in order to reduce the error in the model's results.

The minimum required input metrics, to support risk assessment, required for Dr. Nogueira's estimation model are the following:

a. Efficiency (EF) – The efficiency of the organization can be measured observing the fit between people and their roles [1]. Dr. Nogueira's research indicates that the efficiency of an organization can be directly calculated by computing the ratio of direct time (working and correcting errors) divided by the idle time (time spent without work to do).

b. Requirements Volatility (RV) – Requirements volatility expresses how difficult the requirement elicitation process is. The requirements volatility is obtained by the following formula [1].

Requirements Volatility = Birth Rate Percentage +

Death Rate Percentage

Birth Rate Percentage (BR%) = the percentage of new requirements incorporated in each cycle of the software evolution process as calculated by:

BR% = (New Requirements / Total Requirements) *

100 percent

Death Rate Percentage (DR%) = the percentage of requirements that are dropped by the customer in each cycle of the evolution process as calculated by:

DR% = (Deleted Requirements / Total

Requirements) * 100 percent

c. Complexity (CX) – Complexity has a direct impact on quality because the likelihood that a component fails is directly related to its complexity [1]. The complexity metrics can be determined in two forms: large granular complexity and fine granular complexity. These two forms of complexity can be directly determined from software specifications written in the Prototype System Description Language (PSDL) [3].

Large Granular Complexity (LGC) expresses the relational complexity of the system as a function of the number of operators (O), data streams (D), and types (T)

LGC = O + D + T

Fine Granular Complexity (FGC) expresses the relational complexity of each operator in the system and is a function of the fan-in and fan-out data streams related to the operator [1]. For the purposes of the completed research and our notion of future research, the FGC metric is too specialized; our efforts concentrate on just the representation of the LGC.

FGC = fan-in + fan-out

Software developers can utilize Dr. Nogueira's four models to assess either the development time required to develop a project or determine the associated probability of completing a software project given the project's duration.

3. Previous Validation Research

In this section of the paper we present the results of validation attempts when using Dr. Nogueira's estimation models. The first is a result of the research conducted by Dr. Nogueira in his initial research and supplies data from simulations and comparisons to one project. The second validation endeavor is the results of research conducted on two additional projects [5].

3.1 Dr. Nogueira's Validation

In conducting his research, Dr. Nogueira derived some initial conclusions with the models. The simulations showed that the three risk factors observed during the causal analysis (efficiency, requirements volatility, and complexity) have compound effects over the three parameters of the Weibull distribution [1].

Dr. Nogueira illustrates the results of the models against 16 simulated projects. Each model derives an increasing degree of accuracy based on: metrics from the three risk factors, Weibull cumulative density function, and the derivation of the time.

Models 1-2. Model 1 can be used when the requirements volatility is small. Model 2 considers the three factors (EF, RV, and CX), but neglects the combined effect of EF and RV. Figure 1 illustrates the results of the models which were calculated using 95% of confidence (p=0.95). Note the errors as vertical segments between the estimated and real values.
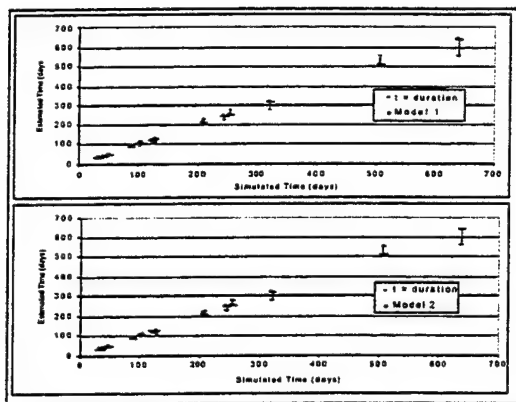
Figure 1. Scatter Plot of Models 1-2

Model 3. Model 3, illustrated in Figure 2, considers the three factors as well as the combined effects of EF and RV. The analysis of variance shows that the samples obtained from the simulations and the samples obtained from the estimates using Model 1, 2 or 3 cannot be statistically differentiated.

Another interesting result is that the errors remain in the range of ±15% for all of the scenarios. This result is interesting if we compare it with the results of COCOMO (±20% in the best cases). Barry Boehm in reference to the validation of COCOMO said, "In terms of our criterion of being able to estimate within 20% of projects actuals, Basic COCOMO accomplishes this with only 25% of the time, Intermediate COCOMO 68% of the time, and Detailed COCOMO 70% of the time." [4].

Model 4. Model 4, Figure 2, can be used for any range of complexity and requirements volatility, and considers the three factors, their combined effects, and the following a priori assumptions:

- A project with 0 LGC will take 0 days
- α, β, and $\gamma > 0$
- If RV increases the $p(x<=t)$ decreases
- If CX increases then $p(x<=t)$ decreases
- If EF increases then $p(x<=t)$ increases
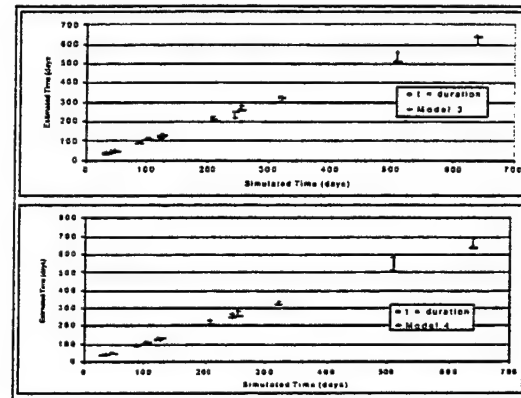
Figure 2. Scatter Plot of Models 3-4

The scatter plot in Figure 2 compares the simulated times versus the estimated times. Most of the errors are overestimations and the duration of the project has no effect over the percentage of error. Model 4 is conservative. The maximum overestimation error was less than 16% and the maximum underestimation was less than 4%.

Model 4 gives a good estimation for projects between 4,000 and 20,000 LGC (128 and 640 KLOC of Ada). The estimation seems to be too optimistic for projects smaller than 1000 LGC but it is quite good for larger projects. To verify the model Dr. Nogueira used a real project consisting of 1836 LGC developed in 1.5 years by the Uruguayan Navy[1]. Model 4 predicts 17 months instead of 18 months, the actual development time.

3.2 Additional Project Validation

Project A [5]. We used Nogueira's Model 4 to calculate the probability of completion curve for the projects. For consistency, we used working days, defined as 22 days per month, the same as used in the original Nogueira model.

The model predicted that the minimum time, in days, necessary to have a probability of completion of 100% is approximately 260 working days. When compared to the actual time it took, which was 336 working days, the model predicted completion sooner. The model predicted 76 working days less, or a 22.6% delta.

$$(1 - (260/336))(100) = 22.6.$$

At this point, with 22.6% variability, we decided to investigate and see what the original

---

[1] SIMTAS a simulator for war gaming with 75,240 lines of code

estimated completion date was from project records. The original estimation was 200 working days, with the project schedule slipping 136 working days for build 3. The developer missed the original completion estimation by 40.5%.

$$(1-(200/336))(100)=40.5.$$

The Nogueira model missed the developer's original estimate by 23.1%

$$(1-(200/260))(100)=23.1$$

Does this mean that the Nogueira model is too optimistic as are most developers' estimates, or is it a better fit? This data point leaves us with an inconclusive position as to the validation of the model against the first project. It appears that there is a difference when using real projects with real data versus simulated project data, and this reflects what the real world is – unpredictable.

Project B [5]. We used Dr. Nogueira's Model 4 to calculate the probability of completion curve for Build 2 using; BR=2.59, DR=3.04, RV=5.63, O=2544, D=4010, T=1003. The model predicted Impossible.

Actual time for build 2 took from 4/24/00 until 7/10/00 or 68 working days at 22 working days a month. We believe this inconsistency is due primarily because the calculation for the LGC count is based on all six Computer Software Configuration Items (CSCI). Core functionality on three CSCIs; CSCI-A, CSCI-B, and CSCI-C had been previously developed and validated. However, the builds during this period, involved addition of functionality to the following CSCIs: CSCI-D, CSCI-E, and CSCI-F. That is, build 2 was modifying only a portion of the total software system code, but the LGC data gives a view of all six CSCIs combined.

The available data was not broken down into separate CSCIs, nor does it, post-mortem, identify the code that was being worked in a previous software release. We cannot fault the developer for not collecting metrics for research concepts that they are not aware of, nor do we believe that this type of data collection is a requirement of CMM level 3.

A finding of this research is the need to adjust the CX when applying the Nogueira model to evolved projects that are developing or enhancing only a portion of their CSCIs.

Additionally, this project did not utilize a lower case tool such as Rational Rose. We believe use of such a tool is essential when attempting to apply the Nogueira formal model, as it provides the capability to collect detailed information, over the software development lifecycle, that can later be extracted and used for input to the Nogueira model metrics.

4. Issues with Dr. Nogueira's Risk Assessment Model

Applying Dr. Nogueira's risk assessment model, in its current form, presents a number of issues that must be resolved before substantial progress can be achieved validating the model's results. The first issue and most notable draw back when using Dr. Nogueira's risk assessment model is limited confidence that the model provides valid results. This is due to three factors: the limited amount of time that the model has been in existence, the model has not been exercised on a wide base of real world projects (completed or on-going), and the fact that the model was developed using simulation techniques. The first factor noted can only be dealt with in the passage of time. However, this research will exploit a unique opportunity to impact the latter two issues.

Although Dr. Nogueira's research shows promise in estimating the associated risk when developing software systems, the model has not been significantly exercised beyond theoretical simulation. Three "real world" projects to date have been applied against the estimation model [1], [5]. It should be noted that all three of these projects were exercised post-mortem. Model validity has not been demonstrated in the context targeted by the model's original design, estimating risk early in a software project's life cycle.

A second issue that exist when using Dr. Nogueira's risk assessment model is the required input metrics. This issue is a double-edged sword. A major attraction to using Dr. Nogueira's model are these metrics. They are determined in a definitive, quantifiable manner and can be derived extremely early in the software development process [1], [6]. However, these metrics are quite unique. Currently, outside of the academic environment, it is not common practice to collect these unique metrics in the required form to utilize Dr. Nogueira's risk assessment model.

In order to establish confidence in the usefulness and accuracy of Dr. Nogueira's risk estimation model, the model must be exercised

against numerous projects. It would be ideal, and perhaps over time, to exercise the model according to it original design; early in the software development cycle. However, the next logical step is to continue to exercise the model in a post-mortem basis. Before this can be accomplished, two things need to happen: First correlations must be determined between Dr. Nogueira's required metrics and metrics that are frequently collected in historical project databases. By establishing metrics correlations, the model can be exercised against an additional project base helping address the second factor of problem one. And second, a method other than the use of PSDL to generate O, D and T metrics counts must be developed. Dr. Nogueira's model was based on using PSDL to automatically scan and generate counts for O, D, and T input to his model. It is unlikely that PSDL was used on any programs that we have post-mortem data on.

The final problem associated with Dr. Nogueira's risk assessment model is the configuration of the Vite'Project simulation. Dr. Nogueira developed the configuration of Vite'Project using Organizational Consultant expert system. Fictitious software engineering organizations were developed to represent the typical software development department. Based on the results of establishing fictitious CMM level 2 and level 3 organizations, the Vite'Project was calibrated. Calibrating the simulation in this manner, could yield different results than calibrating the simulation with actual information derived from real projects. If Dr. Nogueira's model can be verified by reprogramming the Vite'Project configuration this would provide additional assessment to the third factor of problem one.

5. Proposed Research

The proposed research will expand the efforts of the previous validation effort. Figure 3 outlines the research approach.
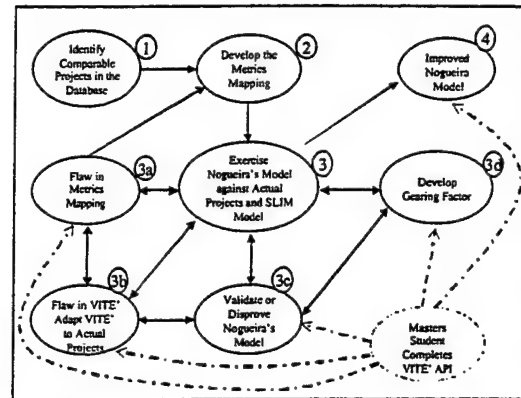
Figure 3. Phases of Research

Phase one: During phase one of the research, post-mortem projects will be identified whose characteristics are similar to the characteristics of the three projects previously exercised against Dr. Nogueira's risk assessment model. This affords the opportunity to begin with a baseline before proceeding to future phases.

Phase two: This is the most challenging phase of the research and we hypothesize that this phase will consume the majority of the available resources. In this phase, detailed analysis is conducted against the available metrics that have been collected on the projects established during phase one. Correlations are determined in the available data against the three metrics that are necessary when utilizing Dr. Nogueira's model. Upon completion of this phase, when a suitable "metric map" has been developed, research can continue to phase three. The intent of the metric map is to provide a common platform to exercise Dr. Nogueira's model using metrics that were not originally collected for this purpose.

Phase three: Once a suitable metric map has been established, research continues by exercising Dr. Nogueira's model against the set of post-mortem projects determined in phase one. This phase is essential to establish confidence in the results produced when using Dr. Nogueira's model. Additionally during this phase, another risk assessment method is introduced, Quantitative Software Management's® (QSM) SLIM, to help in the validation process. Essentially, there will be a comparison of three artifacts: the recorded project performance, the estimated project performance using Dr. Nogueira's model, and

92

the estimated project performance as determined by QSM's SLIM. An assumption during this phase will be the accuracy of QSM's SLIM. Of course, if the expected results are not achieved during this phase, additional research must be performed to determine the cause of the variance.

Phase three (a): One potential cause of the variance observed during phase three could be a flaw in the metric map determined during phase two. Continued research will be conducted to modify the mapping and eventually minimize the chance that the metric map is the source of the deviation.

Phase three (b): Another factor that can influence deviation between the actual project data, Dr. Nogueira's estimation model, and QSM's SLIM estimation model is the original configuration used to establish project scenarios in the Vite'Project. Organizational Consultant expert system was used to establish fictitious software engineering organizations. Research may indicate that reprogramming the Vite'Project with actual information from software development organizations could yield different results in the Vite'Project simulation. This was a fundamental factor in the development of Dr. Nogueira's research. A substantial change in the simulated results could require extensive rework of Dr. Nogueira's model.

Phase three (c): Finally, after exhausting Phases three (a & b), research may lead to examination of Dr. Nogueira's model with closer scrutiny. If deviation continues to present itself when conducting phase three, we may have essentially resort to "ground zero" to establish potential conflicts.

It should be noted that phases three (a, b, & c) should not be considered mutually exclusive. Research could indicate that partial modifications are required in all three sub-phases.

Phase three (d): Dr. Nogueira's risk assessment model is perfectly suited for any evolutionary software process because it follows the same philosophy [1]. Dr. Nogueira presents no hypothesis of the model's validity when the model is exercised outside of this domain. Once phase three is accomplished and confidence has been established against the set of projects determined during phase one, the model can be exercised against additional projects, from different industry sectors and different software development methodologies. This may require the development of what we are calling a "gearing factor". In this research, the use of this term is intended to represent a value that is multiplied by the results determined in Dr. Nogueira's model, adjusting the results for the new domain. In some cases the model may provide suitable results without the use of a gearing factor, other domains and development methodologies may require this adjustment due to the unique nature of the software's development.

Phase four: Phase four of the proposed research is the culmination of all of the proposed research. This phase delivers the improved Nogueira model. A caveat to this phase and all of the sub-phases conducted during phase three is the introduction of the Vite'Project API. This automated tool will improve the statistical significance obtained when utilizing the Vite'Project simulation, greatly increasing the number of simulation runs provided by the simulation.

6. Validation

We propose to validate our research by conducting controlled experiments against post-mortem projects. QSM, founded in 1978 by Larry Putnam, has collected and maintained an extensive database of over 5,000 software projects [7]. Experiments can be conducted, utilizing the available software metrics from QSM's database, that correlate the required metrics in Dr. Nogueira's model. This will afford our research the means to evaluate actual projects against Dr. Nogueira's model.

Another source of validation is obtained by configuring Vite'Project with actual software project development information. As previously mentioned, Vite'Project scenario's were originally established by the creation of fictitious software development organizations. Different results could be derived from simulations configured according to actual projects.

Finally, we propose to increase the statistical significance of Dr. Nogueira's software risk assessment model. We can accomplish this by increasing the simulation runs of each scenario through automation via the Vite' API when available.

7. Conclusion

This research introduces a research plan to validate a formal risk assessment model for software projects based on probabilities and metrics automatically collectable early in the project. The approach enables a project manager to evaluate the probability of success of the

project very early in the life cycle. For more than twenty years the estimation standards (COCOMO 81, COCOMO II, Putnam) have been characterized by a common limitation: the requirements should be frozen in order to make estimations. This promising model removes this important limitation, facing the reality that requirements are inherently variable.

The problem of risk assessment for projects has been treated as unstructured. Research shows, and experiments will prove, a structured method to solve the problem based on metrics automatically collected from the project baselines. This contribution impacts the software engineering state of the art, as well as risk management in general. These metrics measure three risk factors identified in the research: complexity, requirements volatility, and efficiency. The subjectivity issue characteristic of previous research has been eliminated. Any decision-maker will arrive at the same estimates, independently of his or her expertise.

Finally, current research is based on simulations and a small set of real projects. It is desirable to collect and analyze metrics and completion times of a larger set of real software projects to confirm and refine the models. Our research will provide the missing elements from the models, validation, enhancements, and extensions.

References

[1] Nogueira J.C., *A Formal Model for Risk Assessment in Software Projects*. PhD Dissertation. Naval Postgraduate School. Monterey, California. 2000.

[2] The Vite'Project Handbook. Vite©. 1999.

[3] Berzins, V. and Luqi. *Software Engineering with Abstractions*. Addison-Wesley, 1990.

[4] Boehm, B. *Software Engineering Economics*. Prentice Hall, 1981.

[5] Johnson, C. S., Piirainen R. A. *Application of the Nogueira Risk Assessment Model to Real-Time Embedded Software Projects*. Masters Thesis. Naval Postgraduate School. Monterey, California. March 2001.

[6] Nogueira, J.C., Luqi, Bhattacharya, S. A Risk *Assessment Model for Software Prototyping Projects*. Rapid System Prototyping, 2000. RSP 2000. Proceedings. 11th International Workshop on , 2000 Page(s): 28 -33

[7] SLIM MasterPlan User's Guide. QSM® March 2001.

A Unified Approach for the Integration of Distributed Heterogeneous Software Components [1]

Rajeev R. Raje[2][3]   Mikhail Auguston[4][5]   Barrett R. Bryant[4][6]   Andrew M. Olson[2]   Carol Burt[7]

Abstract

Distributed systems are omnipresent these days. Creating efficient and robust software for such systems is a highly complex task. One possible approach to developing distributed software is based on the integration of heterogeneous software components that are scattered across many machines. In this paper, a comprehensive framework that will allow a seamless integration of distributed heterogeneous software components is proposed. This framework involves: a) a meta-model for components and associated hierarchical setup for indicating the contracts and constraints of the components, b) an automatic generation of glues and wrappers, based on a designer's specifications, for achieving interoperability, c) a formal mechanism for precisely describing the meta-model, and d) a formalization of quality of service (QoS) offered by each component and an ensemble of components. A case study from the domain of distributed information filtering is described in the context of this framework.

Keywords: Distributed systems, Formal methods, Glue and Wrapper technology, Quality of Service

1 Introduction

The rapid advances in the processor and networking technologies have changed the computing paradigm from a centralized to a distributed one. This change in paradigm is allowing us to develop distributed computing systems (DCS). DCS appear in many critical domains and are, typically, characterized by: a) a large number of geographically dispersed and interconnected machines, each containing a subset of the required data, b) an open architecture, c) a local autonomy over the hardware and software resources, d) a dynamic system configuration and integration, e) a time-sensitivity of the expected solution, and f) the quality of service with an appropriate notion of compensation. These characteristics make the software design of DCS an extremely difficult task.

One promising approach to the software design of DCS is based on the principles of distributed component computing. Under this paradigm DCS are created by integrating geographically scattered heterogeneous software components. These components constantly discover one another, offer/utilize services, and negotiate the cost and the quality of the services. Such a view provides a scalable solution and hides the underlying heterogeneity.

Various distributed component models, each with strengths and weaknesses, are prevalent and widely used. However, almost a majority of these models have been designed for 'closed' systems, i.e., systems, although distributed in nature, are developed and deployed in a confined setup. In contrast, a direct consequence of the heterogeneity, local autonomy and the open architecture is that the software realization of DCS requires combining components that adhere to different distributed models. This in turn increases the complexity of the design process of DCS. Hence, a comprehensive framework, that provides a seamless access to underlying components and aids in the design of DCS, is needed.

In this paper, one such framework is described. This framework consists of: a) a meta-model for components and associated hierarchical setup for indicating the contracts and constraints of the components, b) an automatic generation of glue and wrappers, based on a designer's specifications, for achieving interoperability, c) a formal mechanism for precisely describing the meta-model, and d) a formalization of the notion of quality of service offered by each component and an ensemble of components. The paper also presents a case study that shows the application of the framework to a specific problem domain.

The rest of the paper is organized as follows. The next section contains a detailed discussion about the meta-model. As an application of the meta model, a case study from the domain of distributed information filtering is presented in the Section 3. Section 4 deals with the formal specification of the meta model, the automated system integration, and evaluation of the approach. Finally, we conclude in Section 5.

2 Component Models and a Meta-model

Many models and projects for the software realization of DCS have been proposed by academia and industry. A few prominent ones are: Java^{TM} Remote Method Invocation (RMI) [16], Common Object Request Broker Architecture (CORBA^{TM}) [16, 20], Distributed Component Object Model (DCOM^{TM}) [11, 16], Web-component model/DOM [10], Pragmatic component web [5], Hadas [6], Infospheres [4], Legion [22], and Globus [21]. Each of these models/projects has strength and weaknesses. Some of these are language-centric and only assume a uniform way of the world (Java); while the others allow a limited interoperability (CORBA – allowing implementations in different languages). Some of these are general-purpose, i.e., not concentrating on any particular application domain (DCOM), while others are specifically tailored to high-performance computing applications (Legion). However, almost all of these models/projects do not assume the presence of other models. Thus, the interoperability which they provide is limited mainly to the underlying hardware platform, operating system and/or implementational languages. Also, there are hardly any models which emphasize the notion of quality of service offered by the components. Projects, such as Agent TCL [8], etc., based on the principles of intelligent agents have imbibed the notion of the quality of service and related compensation. However, the agents are at a higher level of abstraction than components and many of the agent projects/frameworks use one or the other existing distributed-component models at the low-level.

2.1 Why a Meta-model?

Given the above mentioned plethora of component-based models and also noting the fact that components, by their definition, are independent of the implementation language, tools and the execution environment; it is necessary to answer the questions: *why is a meta-model needed for a seamless interoperation of distributed heterogeneous components?* and *how would a meta-model assist in seamlessly integrating distributed heterogeneous software components?* The answer to these question lies in: a) in any organization, software systems undergo changes and evolutions, b) local autonomy is an inherent characteristic of today's geographically (or logically) dispersed organizations, and c) if reliable software needs to be created for a DCS by combining components then the quality of service offered by each component needs to become a central theme of the software development approach.

The consequence of constant evolutions and changes is that there is a need to rapidly create prototypes and experiment with them in an iterative manner. Thus, there is no alternative but to adhere to cyclic (manual or semi-automatic) component-based software development for DCS. However, the solution of decreeing a common COTS environment, in an organization, is against the principle of local autonomy. Hence, the development of a DCS in an organization will, most certainly, require creating an ensemble of heterogeneous components, each adhering to some model. Also, every DCS is designed and developed with a certain goal in mind, and usually that goal is associated with a certain perception of the quality (as expected from the system) and related constraints.

Thus, there is a need for a comprehensive meta-model that will seamlessly encompass existing (and future) heterogeneous components by capturing their necessary aspects, including the quality of service offered by each component and an amalgamation of components.

2.2 Unified Meta-component Model (UMM)

In [17] we have proposed a unified meta-component model (UMM) for global-scale systems. The core parts of the UMM are: *components, service and service guarantees,* and *infrastructure.* The innovative aspects of the UMM are in the structure of these parts and their inter-relations. UMM provides an opportunity to bridge gaps that currently exist in the standards arena. For example, the CORBA Component Model (CCM^{TM}) [13] and Java Enterprise Edition component models (J2EE^{TM}) are consistent, and yet, because of the absence of a formal meta-model, it is difficult during the evolution of each to recognize when the boundaries that maintain the consistency are crossed. Similarly, it has been demonstrated in numerous products that the Component Object Model (COM^{TM}) [18] and CORBA component models are similar (in an abstract sense) enough to allow meaningful bridging. It is, however, not possible to point to a Meta-model that constrains the implementations of these technologies.

For enterprise component solutions, this is an area where significant standards work is now focused. The OMG Meta Object Facility (MOF^{TM}) [14] provides a common meta-model that allows the interchange of models between tools as well as the expression of models in XMI^{TM} (an MOF compliant XML^{TM} (eXtended Markup Language)) [12]. This work allows the generation of interfaces from Unified Modeling Language (UML) [19] models, however, a careful analysis of the resulting interface specifications makes it clear that distribution is not a key factor in the algorithms used. For example, quality of service requirements for performance, scalability and/or security would dictate the use of iterators, the factoring of interfaces to separate "query" and "administrative" operations, and the use of structures and/or objects passed by value. The current standards in this tend to focus on data access with accessors and mutators and relationship transversal. This is acceptable in a single machine environment, but unacceptable for highly distributed communications and collaborations. The recent shift in focus for the Object Management Group to "Model Driven Architecture" (MDA^{TM}) [15] is a recognition that to create mechanized software for the collaboration and bridging of component architectures will require standardization

of Business and Component Meta-Models. The need to support the evolution of component models and to describe the capabilities of the models will be key to realizing the full potential of an E-business economy.

The following sections describe the various aspects of UMM in detail.

2.2.1 Component

In UMM, components are autonomous entities, whose implementations are non-uniform, i.e., each component adheres to some distributed-component model and there is no notion of either a centralized controller or a unified implementational framework. Each component has a state, an identity and a behavior. Thus, all components have well-defined interfaces and private implementations. In addition, each component in UMM has three aspects: 1) a computational aspect, 2) a cooperative aspect, and 3) an auxiliary aspect.

Computational Aspect

The computational aspect reflects the task(s) carried out by each component. It in turn depends upon: a) the objective(s) of the task, b) the techniques used to achieve these objectives, and c) the precise specification of the functionality offered by the component. In DCS, components must be able to 'understand' the functionality of other components. Thus, each component in UMM supports the concept of introspection, by which it will precisely describe its service to other inquiring components. There are various alternatives for a component to indicate its computation – ranging from simple text to formal descriptions. Both these extremes have advantages and drawbacks. UMM takes a mixed approach to indicate the computational aspect of a component – a simple textual part, called *inherent attributes* and a formal precise part, called *functional attributes*.

The functional part is formal and indicates precisely the computation, its associated contracts and the level(s) of service offered by the component. Multi-level contracts for components have been proposed by [2], classifying the contracts into four levels – syntactic, behavioral, concurrency and quality of service (QoS). UMM integrates this multi-level contract concept into the functional part of the computational aspect. As stated earlier, in DCS each component will be offering a service and hence, the level related to the QoS is especially critical in UMM. The QoS depends upon many factors such as, the algorithm used, the execution model, resources required, time, precision and classes of the results obtained. UMM makes an attempt at quantifying the QoS by creating a vocabulary and providing multiple levels of quality, which could be negotiated by the components involved in an interaction. The functional part will also be specified by the creator of the component.

Cooperative Aspect

In UMM, components are always in the process of cooperating with each other. This cooperation may be task-based or greed-based. The cooperative aspect depends on many factors: detection of other components, cost of service, inter-component negotiations, aggregations, duration, mode, and quality. Informally, the cooperative aspect of a component may contain: 1) Expected collaborators – other components that can potentially cooperate with this component, 2) Pre-processing collaborators – other components on which this component depends upon, and 3) Post-processing collaborators – other components that may depend on this component.

Auxiliary Aspect

In addition to computation and cooperation, mobility, security, and fault tolerance are necessary features of DCS. The auxiliary aspect of a component will address these features. In UMM, each component can be potentially mobile. The mobility of the component will be shown as a 'mobility attribute' (a notion similar to the inherent attribute). If a component is mobile, then the mobility attribute will contain the necessary information, such as its implementation details and required execution environment. Similarly, security in DCS is a critical issue. The security attribute of a component will contain the necessary information about its security features. As DCS are prone to frequent failures, full and partial, fault tolerance is critical in these systems. Similar to mobility and security, each component contains fault-tolerant attributes in its auxiliary aspect.

2.2.2 Service and Service Guarantees

The concept of a service is the second part of the UMM. A service could be an intensive computational effort or an access to underlying resources. In DCS, it is natural to have several choices for obtaining a specific service. Thus, each component, in addition to indicating its functionality, must be able to specify the cost and quality of the service offered.

The nature of the service offered by each component is dependent upon the computation performed by that component. In addition to the algorithm used, expected computational effort and resources required, the cost of each service will be decided by the motivation of the owner and the dynamics of supply and demand. In a dynamic environment costs must always be accompanied by the duration for which the costs are valid. As the system dynamics undergo constant changes, the methodologies used to fix the cost of a service will evolve as time progresses, thereby creating a need to indicate the time sensitiveness of the cost. The quality of service is an indication given by an component, on behalf of its owner, about

its confidence to carry out the required services in spite of the constantly changing execution environment and a possibility of partial failures. The techniques used to determine the cost, the time-validity and the quality of a service will depend upon the tasks carried out by the component and the objectives of its owner and will involve principles of distributed decision making.

There are many parameters that a component can use to indicate its quality of service. A few examples are: i) Throughput – number of methods executed per second and classification of methods based on their read/write behaviors, ii) Parallelism constraints – synchronous or asynchronous, iii) Priority, iv) Latency or End-to-End Delay – turn-around time for an invocation, v) Capacity – how many concurrent requests a given component can handle, vi) Availability – indication of the reliability of a component, vii) Ordering constraints – can invocations (asynchronous) be executed out of order by a component, viii) Quality of the result returned – does the component provide a classification or ranking of the result, and ix) Resources available – how many resources (hardware/data) are accessible to the component under consideration and what are the types of resources.

When a component uses certain metrics to indicate its QoS (either all the mentioned criteria or a sub/super set of them), three interesting issues need to be addressed: a) how does the component developer decide these parameters?, b) how does the developer guarantee the advertised QoS during the execution?, and c) when components are collected together as a solution for specific DCS, what happens to the QoS of the combination and how does the combined QoS meet the quality requirements of DCS?

The parameters to be used to describe the QoS of a component are highly context (application) dependent. The proposed approach is to create lists of QoS metrics for common application domains. A few examples of such domains are: scientific computing, multi-media applications, information filtering, and databases. Once such lists are created, they would be used as a template by the component developers while advertising the QoS of their components.

QoS of Components

The issue of guaranteeing a particular QoS, for a component, in an ever changing dynamic DCS is extremely critical; mainly because of external (e.g., policy matters related to resources) and internal (e.g., changes in algorithms) factors that affect a life cycle of a component. In addition, as the software realization of DCS is based on an amalgamation of heterogeneous components, a proper guarantee of a QoS offered by a component effectively decides the QoS of the entire DCS. The quality metrics are expected to vary from one application domain to another and which metrics to select would depend on the intentions of the component developer and the functionality offered by that component. A few examples of such QoS metrics are already mentioned in the previous section. Irrespective of the metrics selected, there is a need for a well-defined mechanism that will assist the developer to achieve the necessary QoS when that component is deployed. Just like any software development process, the process of guaranteeing a certain QoS, as offered by a component, will be an incremental and iterative one, as will be discussed later.

QoS of an Integrated System

In addition to the QoS of individual components, there is a need to achieve a certain QoS for the ensemble of heterogeneous components assembled for a distributed system under discussion. The QoS of such an amalgamation will be decided by the design constraints of the system under construction. However, the integral characteristics of such a system typically cannot be expressed as a function of individual components but as a property of the whole system behavior. Hence, there is a need for a formal model of system behavior, which will integrate the behaviors of each component in the ensemble along with its QoS guarantees.

The proposed approach to address the problem of QoS is as follows. First, build a precise model of systems behavior (event trace notion), provide a programming formalism to describe computations over event traces, and then apply these in order to define different kinds of QoS metrics. Constructive calculations of QoS metrics on a representative set of test cases is one of cornerstones of the proposed iterative approach to system assembly from components meeting user's query specifications.

This approach to the design of a system behavior model assumes that the run time actions performed within the system may be observed as detectable events. Each event corresponding to an action is a time interval, with beginning, end, and duration. Certain attributes could be associated with the event, e.g. program state, source code fragment, time, etc. There are two binary relations defined for the event space: inclusion (one event may be nested within another), and precedence (events may be partially ordered accordingly to the semantics of the system under consideration). Hence, when executed, a system generates an event trace - set of events structured along the relations above. This event trace actually can be considered as a formal behavior model of the system ("lightweight semantics"). This model could be presented as a set of axioms about event trace structure called event grammar [1].

For example, suppose that the entire system execution is represented by an event of type execute-system. It may contain events of the type evaluate-component-A and evaluate-component-B. Event grammar may contain an axiom:
`execute-system:  (evaluate-component-A evaluate-component-B)*`
which states that evaluate-component-A is always followed by the evaluate-component-B event, and these pairs may be repeated zero or more times.

A new concept for specification and validation of target program behavior based on the ideas of event grammars and

computations over program execution traces has been developed, and assertion language mechanisms, including event patterns and aggregate operations over event traces, to specify expected behavior, to describe typical bugs, and to evaluate debugging queries to search for failures (e.g. gathering run time statistics, histories of program variables, etc.) have been created. An event grammar provides a basis for QoS metrics implementation via target program automatic instrumentation. Since the instrumentation is conditional, it does not deteriorate the efficiency of the final version generated code. This mechanism based on independent models of system behavior makes it possible to define QoS metrics as generic trace computations, so that the same metric may be applied to different versions of an assembled system (via automatic instrumentation). To facilitate use of the event grammar model for the assembled system, the event definitions should be consistent through the entire component space. The QoS metrics for components should adhere to this principle. The process proposed in Section 4.4 for assembling a distributed system from components in a distributed network offers a possible approach to achieving this.

2.2.3 Infrastructure

As local autonomy is inherent in open DCS, forcing every component developer to abide by certain rigid rules, although attractive, is doomed to fail. UMM tackles the issue of non-uniformity with the assistance of the *head-hunter* and *Internet Component Broker*. These are responsible for allowing a seamless integration of different component models and sustaining a cooperation among heterogeneous (adhering to different models) components.

Head-hunter Components

The tasks of head-hunters are to detect the presence of new components in the search space, register their functionalities, and attempt at match-making between service producers and consumers. A head-hunter is analogous to a binder or a trader in other models, with one difference – a trader is passive, i.e., the onus of registration is on the foreign components and not on the trader. In contrast, a headhunter is active, i.e., it discovers other components and makes an attempt to register them with itself. There are many approaches possible for the discovery of components. They range from the standard search techniques to broadcasts and multi-casts to selected machines. At a conceptual basis, UMM does not tie itself to a specific approach but during the prototype development a particular approach will be selected for the discovery process. During registration, each component will inform the head hunter about all its aspects. The head hunter will use this information during matching. A component may be registered with multiple head-hunters. Head-hunters may cooperate with each other in order to serve a large number of components. The functionality of head hunters makes it necessary for them to communicate with components belonging to any model, implying that the cooperative aspect of head hunters be universal. Considering the heterogeneous nature of the components, it is conceivable that the software realization of a distributed system will require an ensemble of components adhering to different models. This requires a mediator, the *Internet Component Broker*, that will facilitate cooperation between heterogeneous components.

Internet Component Broker

The Internet Component Broker (ICB) acts as a mediator between two components adhering to different component models. The broker will utilize adapter technology, each adapter component providing translation capabilities for specific component architectures. Thus, a computational aspect of the adapter component will indicate the models for which it provides interoperability. It is expected that brokers will be pervasive in an Internet environment thus providing a seamless integration of disparate components. Adapter components will register with the ICB and while doing so they will indicate their specializations (which component models they can bridge efficiently). During a request from a seeker, the head hunter component will not only search for a provider, but it will also supply the necessary details of an ICB.

The adapter components achieve interoperability using the principles of *wrap* and *glue* technology [9]. A reliable, flexible and cost-effective development of wrap and glue is realized by the automatic generation of glue and wrappers based on component specifications. Wrapper software provides a common message-passing interface for components that frees developers from the error prone tasks of implementing interface and data conversion for individual components. The glue software schedules time-constrained actions and carries out the actual communication between components.

The functionality of the ICB is analogous to that of an object request broker (ORB). The ORB provides the capability to generate the glue and wrappers necessary for objects written in different programming languages to communicate transparently; the ICB provides the capability to generate the glue and wrappers necessary for components implemented in diverse component models (and providing service guarantees) to collaborate across the Internet. An ORB defines language mappings and object adapters. An ICB must provide component mappings and component model adapters. While the ICB conceptually provides the capabilities of existing bridges (COM-CORBA for example), the ICB will provide key features that are unique; it is designed to provide the auxiliary aspects of the Internet – collaboration between autonomous environments, mobility and security. In addition, the UMM includes quality of service and service guarantees. The ICB, in conjunction with head-hunters provide the infrastructure necessary for scalable, reliable, and secure collaborative business using the Internet.

3 A Case Study

In order to explain the UMM and the proposed approach, below a case study from the domain of distributed information filtering is presented. Although the case study uses a specific domain, the principles can be easily extended to other application domains that involve the software realization of a DCS.

3.1 Distributed Information Filtering

It is desired to develop a global information filtering system, in which, users will be interested in receiving selected information, based on their preferences, from scattered repositories. Usually, a filtering task involves contacting the scattered resources, performing an initial search to gather a subset of documents, representing, classifying and presenting based on the user profile. Many different methods are employed for the sub-tasks involved in filtering. Thus, it can be easily envisioned that different components, each employing a different algorithm to perform these sub-tasks, will be scattered across an interconnected system. Each component may belong to a different model, may quote different costs and offer different qualities of service.

Hence, a typical distributed information filtering system consists of the following types of components: a) Domain Component (DC), b) Wrapper Component (WC), c) Representer Component (RC), d) Classifier Component (CC), and e) User Interaction Component (UIC). In addition to these domain-specific components, headhunter components (HC) and the ICB are needed.

All these components, their aspects and characteristics need to be defined using UMM. For the sake of brevity, only the complete description of the domain component (DC) is shown below.

3.2 Domain Component

The domain component is responsible for maintaining a repository of URLs of associated information sources for particular type (e.g., text, structure, sequence) of information that needs filtering.

For example, the inherent attributes might consist of Author (name of the component developer), Version (current version of the component), Date Deployed, Execution Environment Needed and Component Model (e.g., Java-RMI 1.2.2), Validity (e.g., one month from the deployment), Atomic or Complex (indivisible or an amalgamation of other components, e.g. atomic), Registrations (with which headhunters this component is registered, e.g., H1 – www.cs.iupui.edu/h1 and H2 – www.cis.uab.edu/h2).

An informal description of the functional part of a component may contain:

```
1. Computational Task Description -- e.g., searching a selected set of databases over the Internet.
2. Algorithm Used and its Complexity -- Webcrawling and O(n^2), respectively.
3. Alternative Algorithms -- Indexing.
4. Expected Resources (best, average and worst-cases) -- multi-processor, uni-processor (300MHz
with an CPU utilization of 50%), and uni-processor (100MHz with CPU utilization of 99%), respectively.
5. Design Patterns Used (if any) -- Broker.
6. Known Usages  -- for assembling an up-to-date listing containing addresses of known information
repositories for a particular domain.
7. Aliases--  such a component is usually called a Pro-active Agent.
8. Multi-level contracts:
e.g., for a function like List getURLs (Domain inputDomain, Compensation inputCost), the behavioral
contract could specify the pre-condition to be (valid Domain Name and cost), post-condition to be:
if successful (activeClientThreads++ and cost+=inputCost)
else (raise DomainNotKnownException and InvalidCostException)
and the invariant could be (ListOfURLs > 1). Also, for the same function, the concurrency contract
could specify (maximum number of active threads allowed = 50).
```

The cooperation attributes of the domain component may consist of 1) expected collaborators UIC, WC, HC, TC and RC, 2) pre-processing collaborators HC and TC, and 3) post-processing collaborators RC and UIC.

The auxiliary attributes of the domain component are 1) fault-tolerant attributes, e.g., check-pointing versions, 2) security attributes, e.g., simple encryption, and 3) mobility attributes, e.g.. "not mobile."

For the domain component, the QoS parameters may contain 1) number of available URL's, 2) ranking of URL's, and 3) average rate of URL collection.

A component developer may offer several possible levels of QoS, e.g., L1) novice (number of URL's < 50 and no ranking of URL's and average rate of URL collection ≥ 1 week and average latency ≥ 2 minutes), L2) intermediate (number of URL's < 500 and simple ranking of URL's and average rate of URL collection ≥ 3 days and average latency ≥ 1 minute), and L3) expert (number of URL's < 1500 and advanced ranking of URL's and average rate of URL collection ≥ 1 day and average latency ≥ 5 seconds).
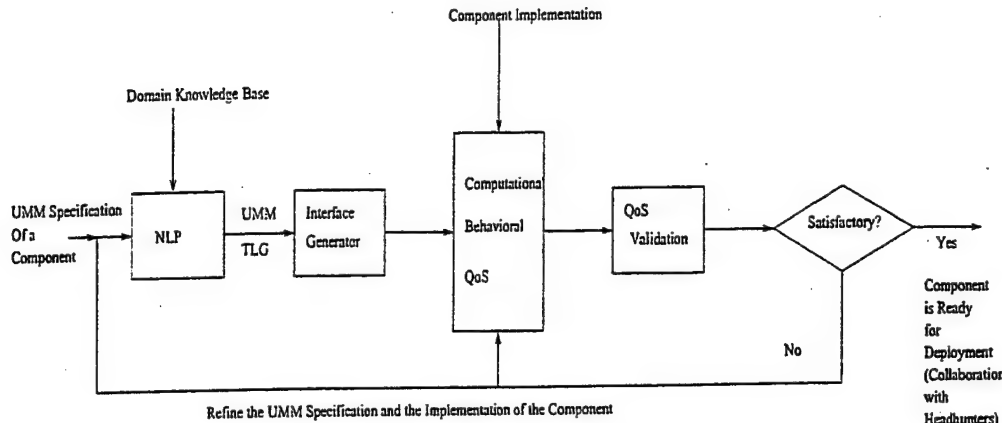
Figure 1: The Component Development and Deployment Process in UMM

The expected compensations for the above levels in terms of the number of URLs could be 1) L1 > 100 and < 200, 2) L2 > 200 and < 400, and 3) L3 > 400 and < 600.

4 Component and System Generation Using UMM Framework

The development of a software solution, using the UMM approach, for a DCS has two levels: a) component level – in this level, different components are created by developers, tested and verified from the point of view of QoS, and then deployed on the network, and b) system level – this level concentrates on assembling a collection of components, each with a specific functionality and QoS, and semi-automatically generates the software solution for the particular DCS under consideration. These two levels and associated processes are described below.

4.1 Component Development and Deployment Process

The component development and deployment process is depicted in Figure 1. As seen in the figure, this process starts with a UMM specification of a component (from a particular domain). This specification is in a natural-language format, as illustrated in the previous section. This informal specification is then refined into a formal specification. The refinement is based upon the theory of Two-Level Grammar (TLG) natural language specifications [3, 23], and is achieved by the use of conventional natural language processing techniques (e.g. see [7]) and a domain (such as information filtering) knowledge base. TLG specifications allow for the generation of the interface (possibly multi-level) for a component. This interface incorporates all the aspects of the component, as required by the UMM. The developer provides the necessary implementation for the computational, behavioral, and QoS methods. This process is followed by the QoS validation. If the results are satisfactory (as required by the QoS criteria) then the component is deployed on the network and eventually, it is discovered by one or more headhunters. If the QoS constraints are not met then the developer refines the UMM specification and/or the implementation and the cycle repeats.

4.2 Formal Specification of Components in UMM

Since the UMM specifications are informally indicated in a natural language like style, our approach is to translate this natural language specification into a more formal specification using TLG. TLG is a formal notation based upon natural language and the functional, logic, and object-oriented programming paradigms. The name "two-level" in Two-Level Grammar comes from the fact that TLG consists of two context-free grammars, one corresponding to a set of type declarations and the other a set of function definitions operating on those types. These type and function definitions are incorporated into a class which allows for new types to be created.

The type declarations of a TLG program define the domains of the functions and allow strong typing of identifiers used in the function definitions. On the other hand, function definitions may be given without precisely defined domains for a more flexible specification approach. This framework consists of a knowledge-base which establishes a context for the natural language text to be used in the specification under a particular domain model, in this case information filtering. This allows the TLG to be translated into internal representations such as predicate logic, the natural representation for TLG, event grammars, or multi-level Java interfaces taking the form of the UMM specification template. For the case

101

study, we may use a TLG class to describe the component structure and functionality as elaborated in the following subsections.

4.2.1 Component Structure Specification

Syntactically, TLG type declarations are similar to those in other languages. Types are capitalized whereas constants begin with lower case letters. The usual primitive types, such as Integer, Float, Boolean, and String are present as are list constructors based upon regular expression notation, e.g. {X}* and {X}+ mean 0 or more and 1 or more occurrences of X, respectively.

The types of the domain component in our information filtering system are defined in the following way in TLG.

```
Component :: DomainComponent; WrapperComponent; RepresentationComponent; ClassificationComponent;
  UserInteractionComponent; HeadhunterComponent; ICB.
DomainComponent :: Name, InformalDescription, Attributes, Service.
Name :: dc.
Attributes :: ComputationalAttributes, CooperationAttributes, AuxiliaryAttributes.
ComputationalAttributes :: InherentAttributes, FunctionalAttributes.
InherentAttributes :: Author, Version, DateDeployed, ExecutionEnvironment,
  ComponentModel, Validity, Structure, Registrations.
FunctionalAttributes :: TaskDescription, AlgorithmAndComplexity,
  Alternatives, Resources, DesignPatterns, Usages, Aliases, FunctionsAndContracts.
AlgorithmAndComplexity :: webcrawling, n^2; ....
Alternatives :: {AlgorithmAndComplexity}*.
Resource :: Architecture, Speed, Load.
Architecture :: uni-processor; multi-processor.
Speed :: Integer.
Load :: Integer.
DesignPatterns :: broker; ....
Aliases :: pro-active agent; ....
FunctionAndContract :: Function, BehavioralContract, ConcurrencyContract.
Function :: ....
BehavioralContract :: Precondition, Invariant, Postcondition.
ConcurrencyContract :: single threaded; maximum number of active threads allowed = Integer; ....
CooperationAttributes :: ExpectedCollaborators, PreprocessingCollaborators, PostprocessingCollaborators.
ExpectedCollaborator :: uic; wc; hc; tc; rc.
PreprocessingCollaborator :: hc; tc.
PostprocessingCollaborator :: rc; uic.
AuxiliaryAttribute :: FaultTolerantAttribute; SecurityAttribute; MobilityAttribute.
FaultTolerantAttribute :: check-pointing versions; ....
SecurityAttribute :: simple encryption; ....
MobilityAttribute :: mobile; not mobile.
Service :: ExecutionRate, ParallelismConstraint, Priority, Latency, Capacity, Availability,
  OrderingConstraints, QualityOfResultsReturned, ResourcesAvailable, ....
ExecutionRate :: Float.
ParallelismConstraint :: synchronous; asynchronous.
Priority :: Integer.
Latency :: AverageRateOfURLCollection.
AverageRateOfURLCollection :: Float.
Capacity :: NumberOfAvailableURLs.
NumberOfAvailableURLs :: Integer.
Availability :: Float.
OrderingConstraint :: Boolean.
QualityOfResultsReturned :: {URL}+.
ResourcesAvailable :: HardwareResources, SoftwareResources.
HardwareResources :: ....
SoftwareResources :: ....
```

The remaining components (e.g., wrapper, representation, etc.) may be described in a similar manner. All domains not specified explicitly in the above example are assumed to be of type String, with the exception of Function which may take the form of an interface definition in a programming language such as Java. Using standard natural language processing techniques [7], the UMM specification may be automatically refined into this TLG specification, with user assistance as

needed to clarify ambiguities. The process is facilitated by the presence of a knowledge base which understands the domain of information filtering from the point of view of vocabulary which may be used in making the original UMM specification.

4.2.2 Component Functionality Specification

The second level of the TLG specification is for function declarations. These resemble logical rules in logic programming with variables coming from the domains established in the type declarations. For the Domain Component example, the levels of Quality of Service may be specified as follows.

```
number of urls : size of QualityOfResultsReturned.
average latency : ...
no ranking of urls : ...
simple ranking of urls : ...
advanced ranking of urls : ...
average latency : ...
qos level 1 is novice : number of urls < 50, no ranking of urls,
   AverageRateofURLCollection >= 1 week, average latency >= 2 minutes.
qos level 2 is intermediate : number of urls < 500, simple ranking of urls,
   AverageRateofURLCollection >= 3 days, average latency >= 1 minute.
qos level 3 is expert : number of urls < 1500, advanced ranking of urls,
   AverageRateofURLCollection >= 1 day, average latency >= 5 seconds.
```

Each rule defines how the particular entity is to be computed. As these rules are normally part of a class definition encapsulating a corresponding set of type declarations, each rule has access to the data specified in the type declarations. These natural language like rules may be further refined into a more formal specification, e.g. using event grammars.

4.3 QoS Guarantee of a Domain Component

For the case study, the event grammar to describe the system behavior is given below. The first part is the set of type definitions and the second part is the description of computations over event traces implementing different QoS metrics.

```
exec_syst :: (request_sent | response_received)*
response_received :: (URL_detected | failed)
```

These type definitions describe the types of events which may occur as the system executes. The computations over these events include verification that the number of URL's detected is less than 50 and also the latency (e.g., for all requests for URL's, every response received occurs within 10 units of time). id is an event attribute which associates a unique identifier between query attributes and corresponding responses. Both of these metrics yield Boolean values.

```
CARD [URL_detected from exec_syst] < 50

Forall x : request_sent from exec_syst
  Exists y : response_received from exec_syst
    id (x) = id (y) & begin_time (y) - end_time (x) < 10
```

4.4 Automated System Generation and Evaluation based on QoS

In general, different developers will provide on the Internet a variety of possibly heterogeneous components oriented towards a specific problem domain. Once all the components necessary for implementing a specified distributed system are available, then the task is to assemble them. Figure 2 shows a process to accomplish this. The developer of the desired distributed system presents to this process a system query, in a structured form of natural language, that describes the required characteristics of the distributed system. For example, such a query might be a request to assemble an information filtering system. The natural language processor (NLP) processes the query. It does this aided by the domain knowledge (such as key concepts in the filtering domain) and a knowledge-base containing the UMM description (in the form of a TLG) of the components for that domain. The result is a formal UMM specification that will be used by headhunters for component searches and as an input to the system assembly step. This formal UMM specification will be a basis for generating a set of test cases to determine whether or not an assembly satisfies the desired QoS. The framework, with the help of the infrastructure described in Section 2.2.3, collects a set of potential components for that domain, each of which meets the QoS requirement specified by the developer. From these, the developer, or a program acting as a proxy of the developer, selects some components. These components along with the component broker and appropriate adapters (if needed) form a software implementation of the distributed system. Next this implementation is tested using event traces and the set of test cases to verify that it meets the desired QoS criteria. If it does not, it is discarded. After that, another implementation is chosen from the component collection. This process is repeated until an optimal (with respect to the QoS) implementation is found, or until the collection is exhausted. In the latter case, the process may request additional
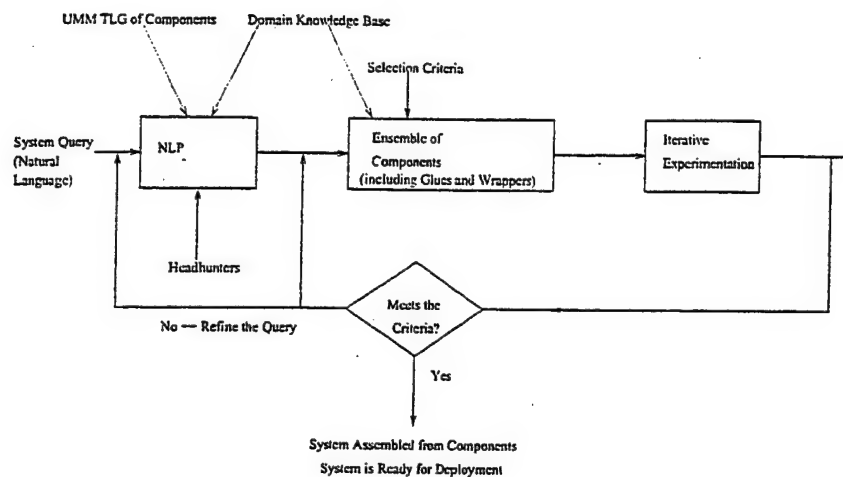
Figure 2: The Iterative System Integration Process in UMM

components or it may attempt to refine the query by adding more information about the desired solution from the problem domain. Once a satisfactory implementation is found, it is ready for deployment.

5   Conclusion

This paper has presented a framework that allows an interoperation of heterogeneous and distributed software components. The software solutions for future DCS will require either automatic or semi-automatic integration of software components, while abiding with the QoS constraints advertised by each component and the collection of components. The result of using UMM and the associated tools is a semi-automatic construction of a distributed system. Glue and wrapper technology allows a seamless integration of heterogeneous components and the formal specification of all aspects of each component will eliminate ambiguity while detecting and using these components. The UMM does not consider network failures or other considerations related to the hardware infrastructure, however, these could be integrated into the QoS level of components. The UMM approach to validating QoS is to use event grammar to calculate QoS metrics over run-time behavior. The QoS metrics are then used as a criteria for an iterative process of assembling the resulting system as shown in Section 4.4. UMM also provides an opportunity to bridge gaps that currently exist in the standards arena. Although, the paper has only presented a case study from the domain of distributed information filtering, the principles of UMM may be applied to other distributed application domains. Future work includes refinement of the UMM feature thesaurus and methods for translating UMM specifications into Two-Level Grammar, refining the head-hunter mechanism, developing Quality of Service metrics for components and systems, and development of generation mechanisms for domain-specific component reuse.

References

[1] Auguston, M. A Language for Debugging Automation. In *Proceedings of 6th International Conference on Software Engineering and Knowledge Engineering*, pages 108–115, 1994.

[2] Beugnard, A., Jezequel, J., Plouzeau, N. and Watkins, D. Making Components Contract Aware. *IEEE Computer*, 32(7):38-45, July 1999.

[3] Barrett R. Bryant. Object-Oriented Natural Language Requirements Specification. In *Proceedings of ACSC 2000, the 23rd Australasian Computer Science Conference, January 31-February 4, 2000, Canberra, Australia*, pages 24–30, January 2000.

[4] California Institute of Technology. *Caltech Infospheres On-line Documentation*, URL:- http://www.infospheres.caltech.edu/, 1998.

[5] Fox, G. The Document Object Model Universal Access Other Objects CORBA XML Jini JavaScript etc. http://www.npac.syr.edu/users/gcf/msrcobjectsapril99, 1999.

[6] Israel, B. and Kaiser, G. Coordinating Distributed Components Over the Internet. *IEEE Internet Computing*, pages 83–86, 2(2), 1998.

[7] Jurafsky, D. and Martin, J. H. *Speech and Language Processing*. Prentice Hall, 2000.

[8] Kotz, D., Gray, R., Nog, S., Rus, D., Chawla, S. and Cybenko, G. Agent TCL: Targetting the Needs of Mobile Computers. *IEEE Internet Computing*, pages 58–67, 1(4), 1997.

[9] Luqi, Berzins, V., Ge, J., Shing, M., Auguston, M., Bryant, B. R. and Kin, B. K. DCAPS - Architecture for Distributed Computer Aided Prototyping System. In *Proceedings of RSP 2001, the 12th International Workshop on Rapid System Prototyping*, 2001.

[10] Manola, F. Technologies for a Web Object Model. *IEEE Internet Computing*, 3(1):38–47, January-February 1999.

[11] Microsoft Corporation. *DCOM Specifications, URL:- http://www.microsoft.com/oledev/olecom*, 1998.

[12] Object Management Group. XML Metadata Interchange. Technical report, Object Management Group Document No. ad/98-10-05, October 1998.

[13] Object Management Group. CORBA Components. Technical report, Object Management Group TC Document orbos/99-02-05, March 1999.

[14] Object Management Group. Meta Object Facility (MOF) Specification, Version 1.3. Technical report, Object Management Group, March 2000.

[15] Object Management Group. Model Driven Architecture: A Technical Perspective. Technical report, Object Management Group Document No. ab/2001-02-01, February 2001.

[16] Orfali R, and Harkey, D. *Client/Server Programming with JAVA and CORBA*. John Wiley & Sons, Inc., 1997.

[17] Raje, R. UMM: Unified Meta-object Model for Open Distributed Systems. In *Proceedings of the fourth IEEE International Conference on Algorithms and Architecture for Parallel Processing (ICA3PP'2000)*, 2000.

[18] Rogerson, D. *Inside COM*. Microsoft Press, 1996.

[19] Rumbaugh, J., Jacobson, I. and Booch, G. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998.

[20] Siegel, J. *CORBA Fundamentals and Programming*. John Wiley & Sons, Inc., 1996.

[21] The Globus Project. *Globus Website, URL:- http://www.globus.org/*, 2000.

[22] University of Virginia. *Legion Project, URL:- http://www.cs.virginia.edu/ legion*, 1999.

[23] van Wijngaarden, A. Orthogonal Design and Description of a Formal Language. Technical report, Mathematisch Centrum, Amsterdam, 1965.

Optimization of Distributed Object-Oriented Servers

William J. Ray

SPAWAR Systems Center, (619) 553-4150, ray@spawar.navy.mil

Valdis Berzins

Naval Postgraduate School, (831) 656-2610, berzins@cs.nps.navy.mil

Abstract - This paper presents a method for deploying distributed object servers to optimize client response time. Object-Oriented (OO) computing is fast becoming the de-facto standard for software development. Distributed OO systems can consist of multiple object servers and client applications on a network of computers, as opposed to a single large centralized object server. Optimal deployment strategies for object servers change due to modifications in object servers, client applications, operational missions and changes in various other aspects of the environment.

As multiple distributed object servers replace large centralized servers, there is a growing need to optimize the deployment of object servers to best serve the end user's changing needs. A method that automatically generates object server deployment strategies would allow users to take full advantage of their network of computers.

States of the art load balancing techniques schedule a given number of independent tasks on a set of machines. However, object servers do not have independent tasks: all methods in an object are related. Also, the number of times a method is called is usually dependent on interactions with end users.

The proposed method profiles object servers, client applications, user inputs and network resources. These profiles determine a system of non-linear equations that is solved to produce an optimal deployment strategy.

Keywords: Distributed Object, Load Balancing, Client Response Time, Optimization, Server Deployment and Software Engineering.

1. INTRODUCTION
The future of computing is heading for a universe of distributed object servers. The evolution of object servers to distributed object servers will parallel the evolution of the relational databases. Over time, object servers will provide functionality to more client applications than their original applications, just as relational databases were used by more applications than the original application. In both cases, systems optimized for the original application may not perform well for the new applications. Tools that allow a programmer to model an object and easily create object servers with all the necessary infrastructure code needed to work as a distributed object server are available [12]. This will lead to an explosion in the number of object servers available to client applications.

A user's network of computers will change frequently. Object servers, applications, hardware and user preferences will be in a constant state of flux. No static deployment strategy can adequately take advantage of the assets accessible on the network in such an environment.

No system can accurately predict user interaction with a system. Two separate users performing the same job will interact with a system differently. The same user may interact differently while performing the same job at different times. For these reasons and combinatorial explosion problems, an adaptive software engineering approach is proposed instead of a traditional computer science approach.

Most deployment strategies today are dictated by the system engineer's view of how the systems will be utilized. Of course, the system engineer doesn't revisit these strategies every time hardware, software or user interactions change. The goal is to allow the user to update hardware and usage profiles. Software developers would supply new profiles when their code changes. Any time a profile is updated, the model would be run and an automated reconfiguration of the object server deployment could occur. In most cases, the frequency of change will be greatest in the hardware and usage pattern profiles. Since many of these changes can take place without the knowledge of a system engineer or the budget to employ one, a method that allows the users to update these profiles and initiate the reconfiguration is desired.

2. PREVIOUS WORK

There has been little work on deployment strategies for distributed object servers. The closest relevant research is in the fields of load balancing, client/server performance and distributed computing. Most state of the art load balancing techniques address scheduling of given set of tasks on a set of given machines. Some techniques only deal with tasks that are independent. Others deal with dependent tasks that are usually linked together by temporal logic and mutual exclusion constraints.

Object servers do not have independent tasks. All methods in all object types in a single object server are related at least by locality and more often by the interaction between the object types. Also, the number of times a method is called is not given, but rather depends on undetermined interactions with end users, very much like the situation in client/server performance research. We propose a system that enables optimization of object server deployment to meet changing needs.

3. CURRENT PRACTICES

Because of the difficulty in producing the infrastructure code necessary to support distributed object computing, many developers produce huge monolithic object servers [11]. A powerful machine is usually needed to adequately handle this server and successful applications that experience large increases in the number of users may outgrow the capabilities of the fastest available single machine. With automated code-generation tools, these servers will be much easier to produce and reconfigure [12]. This allows servers to be partitioned by allocating unrelated or loosely related objects types to different physical servers that can be deployed across the network to take advantage of the available assets. By taking advantage of all the assets on the network, faster response times can be achieved [11].

Loosely related object types are defined as object types that contain associations to other object types. When these object types reside in different physical object servers, the result is an object server that calls on other object servers. A server that calls other servers is a complex server [1].

Many networks of computers are installed with a single purpose in mind. Over time, these networks support an evolving set of tasks. Even though the original role the network played can change dramatically, rarely does a single system engineer revisit the deployment strategy for the entire system. What a user ends up with is usually the product of multiple system engineers' choices made based on the latest incremental changes without regard for the system as a whole and interactions among its roles. It is infeasible, because of cost, to hire a system engineer to re-assess the whole system every time a change occurs. In the end, the user is left with a system whose deployment strategy borders on randomness.

4. OPTIMIZATION OF DISTRIBUTED OBJECT-ORIENTED SYSTEMS

The goal of this paper is to describe a method that can generate distributed object oriented server deployment architectures to take advantage of network resources for the purpose of reducing average client response time. A system that carries out this method must be able to reason about deployment strategies of loosely related objects. The proposed system maps all of these profiles into equations to minimize average client response time.

Average client response time was chosen as the optimization criteria over others. In this paper, the goal was to be user centric. Criteria that focused on maximizing machine utility were not germane. Average client response time was chosen over minimizing the maximum response time of one call because the method takes into account the entire usage profile.

4.1 Optimization Model

The equations that need to be solved will minimize the sum of all of the response times for a given call pattern over a given time interval. Since we want to allow the user the freedom to run client applications from anywhere on the network, we will ignore all processing on the client machines and all network delay between client machines and server machines. The only factors we will consider for optimizing our server deployment are the processing on the object server and the network delay between complex object servers. Therefore, the objective function that we wish to minimize is:

$$Minimize \left[\sum_{n=0}^{N} \sum_{m=0}^{M} \frac{a_{nm} * R_n * S_{norm}}{S_m} + \sum_{i=0}^{N} \sum_{j=0}^{N} \frac{B_{ij}}{Q_{ij}} \right]$$

subject to the following four constraints:

1. Object Servers cannot be split across machines.

$$a_{nm} \quad = \quad \begin{array}{l} 1, \text{ iff server n is running on machine m} \\ 0, \text{ otherwise} \end{array}$$

2. Each Server can run on only one machine [no multiple instances of the same server.

$$\forall n \left[\sum_{m=0}^{M} a_{nm} \equiv 1 \right]$$

3. RAM usage by the object servers cannot pass a set threshold on each machine.

$$\forall m \left[\sum_{n=0}^{N} a_{nm} * V_n \leq T_m * U \right]$$

4. CPU time on a given machine cannot surpass the corresponding real time interval.

$$\forall m \left[\sum_{n=0}^{N} \frac{a_{nm} * R_n * S_{norm}}{S_m} \leq C \right]$$

where

N	= Number of object servers
M	= Number of physical machines
a_{nm}	= server n is running on machine m
R_n	= Normalized machine load of server n (seconds, s)
S_{norm}	= Speed of the normalizing machine (MHz)
S_m	= Speed of machine m (MHz)
B_{ij}	= Data sent between server i to server j (bits, b)
Q_{ij}	= Network Speed between server i to server j (bps)
T_m	= Physical RAM on machine m (bits, b)
V_n	= Memory allocated by server n (bits, b)
U	= Multiple to limit RAM utilization [0.1,3.0]
C	= Time Interval [seconds, s]

NOTE: All terms are fixed either by measurement or input except for a_{nm}. The model varies all possible combinations for a_{nm} and finds the minimum based on the above objective function and constraints.

4.2 Evolution

Over time, a collection of hardware, software and user requirements will change in a given environment. Common hardware changes consist of adding new computers, removing old computers, upgrading CPUs, modifying RAM and modifying network bandwidth capacity. Each of these hardware changes will produce an event that would trigger the system to re-evaluate its deployment strategy.

Software can also be quite dynamic in nature. New object servers and applications can appear. Old ones can be removed. Existing object schemata and methods can be changed. Each of these changes would trigger an event to re-evaluate the deployment strategy.

4.3 Loosely Related Objects
Not all objects types that are related must necessarily be contained in a single object server. There is a point where the performance of the system would improve by moving the object type into a different server. This is usually the case when none of the application code exercises an inter-server method call or exercises it only very rarely. Large message sizes and slow network speeds will push for related object types to be co-located. The approach will be able to reason about not only deploying object servers, but also recommend the schema supported by these object servers.

4.4 Priority Setting
User requirements can also be in a state of flux. Most computer systems are used to support multiple jobs. Business-hour requirements can differ greatly from after-hours computational requirements. A developer's network of computers can support multiple projects, but may need to be optimized for a single project for demonstrations. In the military, the operational mission being supported can change significantly. For example, a set of distributed object servers could be used to support many applications aboard a ship. These applications could handle such tasks as Anti-Submarine Warfare (ASW), Anti-Surface Warfare (ASUW), Anti-Air Warfare (AAW), Electronic Warfare (EW), humanitarian missions and rescue missions. The relative computational activity of these applications could differ significantly on different missions of the ship.

Optimizing a system of object servers for all possible roles would not be optimal when the system is only performing a couple of missions at a time. By profiling each role, the user could choose to re-optimize his deployment to decrease the response time when user chosen roles change. In this way, the user could tune his system to give peak performance for the task he is currently trying to perform.

4.5 Profiles
The tricky part is to figure out what elements are needed in the different profiles, how to map these profiles into equations and then model how these profiles interact with each other. The more complex the modeling of the hardware becomes the more computationally intensive the approach will become. Initially we demonstrate an approach with rather simplistic profiles to demonstrate its capabilities.

4.5.1 Hardware Profiles
The aspects being modeled in the hardware profiles include characteristics of each computer such as CPU speed and physical RAM size. The hardware profile also models the network speed between each computer. Current hardware profiles do not directly support multi-processor computers, but they could be modeled as groups of separate nodes with very high "network speeds" between them.

4.5.2 Object Server Profiles
Object servers need to be profiled for metrics associated with each method call in each object. The computational time of each method call should be captured and normalized to a specific hardware architecture. Since object servers ideally run continuously, the RAM of the object server must also be measured and summarized. The hardware profile and the object server profile is sufficient to optimize the server deployment for the case where all the functionality contained in all the objects is of equal value to the user. Metrics can be collected easily with a small client application that exercises each method call and records the data. Thus, actual implementation code for the application isn't needed to estimate the object server profiles.

4.5.3 Client Application Profiles
Ideally, client applications would be delivered with their profiles. If the code is available, then the source can be parsed to find all possible object invocations. Since exact frequencies of method calls are not algorithmically computable in the general case, measurement is necessary to reliably estimate frequencies of calls. The system must allow a user to create typical scenarios and record the method calls that occur in the scenario. This could be done by simulation or monitoring calls to the object servers when the system is in a training mode. The plus side to this method is that the user could represent more complex tasks involving many user interactions in a single profile. Numerous tools exist for complex event processing in a distributed system [5, 6].

4.5.4   User Profiles

User profiles or roles indicate how a user interacts with the system over a given period of time. In simplistic terms, it is like keeping track of how many times each button is selected over a given time interval. Average button push rates can be expressed as number of events per second. The user can collect this data manually or automatically by the system with audit trails. Multiple roles can exist for each user. The user could then select a set of roles and have the system come up with an optimal deployment strategy to meet these criteria.

4.6 Profile Mappings

In order to compute the optimal deployment strategy given a set of profiles, one needs to map these profiles into equations that can be solved for minimum response time. To illustrate the mappings, we present an example. The example consists of three machines, three object servers and three client applications. The method demonstrates the differences in deployment for a system tuned to a users-specific role. Table 1 shows the profile for the computer hardware available.

Table 1. Machine profile for example.

MACHINE	RAM (bits)	CPU Speed (MHz)
SIX	512,000,000 = 64MB	600
BR733	1,024,000,000 = 128MB	733
GIGA	1,024,000,000 = 128MB	1000

Table 2 shows the network bandwidth available to communicate from each machine to the other. In this example, the machines will have equal bandwidth between machines as is the case when all servers are running on the same local LAN. The speed of communications between servers on the same machine is more difficult to predict. These speeds usually lie in the interval bounded by the speed of the machines back plane and the speed of the network. It is dependent on the operating system, implementation of the middleware, and other factors. For this example, we assume that intra-machine communication is twice as fast as inter-machine communication. In the absence of measurements, the system can be run with best and worst case scenarios by specifying the boundary values identified above.

Table 2. Network speed.

Machine to Machine Speed (bps)	SIX	BR733	GIGA
SIX	200,000,000	100,000,000	100,000,000
BR733	100,000,000	200,000,000	100,000,000
GIGA	100,000,000	100,000,000	200,000,000

Besides the hardware profiles, we need to have the server profiles. Table three lists each server's RAM requirements.

Table 3. Server RAM requirements.

SERVER	RAM Required (bits)
A	352,000,000 = 44MB
B	480,000,000 = 60MB
C	528,000,000 = 66MB

Additional parts of the object server are the timing of each individual method call available in each server and a list of complex method calls. All of these measurements were taken on a single machine to normalize the values. In this example, server A has one four methods, server B has two methods, and server C has three methods.

Table 4. Normalized Server Loads.

SERVER	Method	CPU time (s)	Average Size of Message (b)
A	1	0.5796	112000
A	2	2.6203	18400
A	3	1.18175	44800
A	4	2.0264	176000
B	1	1.76655	4000000
B	2	3.70085	2720000

C	1	3.0043	320000
C	2	4.8040	4000000
C	3	0.48815	400000

A complex method call is a method call that calls another object server. These method calls require special handling in measuring their load on the host server and in the objective function for optimizing the system. Table 5 lists the complex method calls in this example.

Table 5: Complex Method Calls

Complex Method	Exterior Calls
B.2	C.1

The last information needed to optimize the system is information about the applications and the users. This step adds roles to the list of profiles for the system to optimize. These roles have more realistic use patterns for the different jobs a user would actually perform on the system. For this example, we will have three client applications with two buttons, nine buttons and three buttons respectively.

Let's assume that there are three different roles the network of computers supports for the user and the following is the use pattern shown in Table 6, and that the buttons call the following server methods shown in Table 7. Method calls that appear in italics in Tables 7 and 8 are complex method calls. They appear in italics to remind us that these methods require special handling when figuring out the objective function.

Table 6. Roles.

ROLE	CALL PATTERN (observation interval is 990 seconds)
Role 1	50 C1.B1 + 1 C1.B2 + 1 C2.B1 + 1 C2.B6
Role 2	10 C1.B1 + 40 C1.B2 + 24 C3.B2
Role 3	50 C2.B5 + 10 C2.B9 + 30 C2.B3 + 1 C2.B2 + 1 C3.B2

Table 7. User interface calls.

Button	Methods Called
C1.B1	A.1
C1.B2	A.2 + B.1
C2.B1	C.1 + C.2
C2.B2	C.3
C2.B3	C.2
C2.B4	C.3
C2.B5	A.1 + *B.2*
C2.B6	*B.2*
C2.B7	A.4
C2.B8	C.3 + A.3
C2.B9	A.1 + A.2 + A.3 + *B.2*
C3.B1	C.1
C3.B2	B.1 + *B.2*
C3.B3	C.2

By substituting the user interface calls into the roles matrix, we get an objective function for optimizing the system shown in Table 8. All other method calls will be ignored.

Table 8. Roles to server calls.

ROLE	Methods Called in Role
Role 1	50 * (A.1) + 1 * (A.2 + B.1) + 1 * (C.1 + C.2) + 1 * (*B.2*)
Role 2	10 * (A.1) + 40 * (A.2 + B.1) + 24 * (B.1 + *B.2*)
Role 3	50 * (A.1 + *B.2*) + 10 * (A.1 + A.2 + A.3 + *B.2*) + 30 * (C.2) + 1 * (C.3) + 1 * (B.1 + *B.2*)

4.6.1   Filling in the Equation for Role 1

Role 1 consists of 50 C1.B1 calls, one C1.B2 call, one C2.B1 call, and one C2.B6 call. The first step is to convert all of the button calls into method calls by substituting the values for the calls from Table 4.

50 [A.1] + 1 [A.2 + B.1] + 1 [C.1 + C.2] + 1 *[B.2]* =

50 [A.1] + 1 [A.2 + B.1] + 1 [C.1 + C.2] + 1 [B.2 + C.1] =

50 A.1 + A.2 + B.1 + C.1 + C.2 + B.2 + C.1 =

50 A.1 + A.2 + B.1 + B.2 + 2 C.1 + C.2

This leads to the following values for the array R for the optimization equation.

R (A) = 50 [A.1 values for CPU] + 1 [A.2 value for CPU]

$\quad\quad\quad$ = 50 [579.6] + 1 [2620.3]

$\quad\quad\quad$ = 31600.3

R (B) = 1 [B.1 values for CPU] + 1 [B.2 value for CPU]

$\quad\quad\quad$ = 1 [1766.55] + 1 [3700.85]

$\quad\quad\quad$ = 5467.4

R (C) = 2 [C.1 values for CPU] + 1 [C.2 value for CPU]

$\quad\quad\quad$ = 2 [3004.3] + 1 [4804.0]

$\quad\quad\quad$ = 10812.6

There is only one italicized method call prior to substitution, so there is only one network value to deal with.

BITS[B,C] = 1 [*B.2* message in bits]

$\quad\quad\quad$ = 320000

4.6.2   Filling in the Equation for Role 2

Using the same approach as in 4.6.1, we get the following for Role 2:

R (A) = 110608

R (B) = 201879.6

R (C) = 72103.2

There is only one italicized method call prior to substitution, so there is only one network value to deal with. However, it is called 24 times.

BITS[B,C] = 24 [*B.2* message in bits]

$\quad\quad\quad$ = 24 [320000]

$\quad\quad\quad$ = 7680000

4.6.3   Filling in the Equation for Role 3

R (A) = 72796.5

R (B) = 227518.4

R (C) = 327870.45

BITS[B,C] = 19520000

4.7 Model Solutions

All of the information above is run through a LINGO model that varies the location of the object servers on the different machines to find the a solution set that minimizes the value of the objective function.   The model prompts the user for inputs bandwidth, RAM percentage and computational time limitations. Changing any of these variables will lead to different model outputs [10].

4.8 Model outputs

This method outputs the following deployment strategies for the different roles when setting different RAM limits and keeping all other variables the same as in the last example. Solving the optimization problem defined in section 4.1 with the parameter values determined in section 4.6 derives these results.

Table 9. Single user deployment strategies for different roles. RAM limit set to 1.5.

Machine	Role 1 (user)	Role 2 (1 user)	Role 3 (1 user)
SIX	None	None	None
BR733	None	None	None
GIGA	A, B, C	A, B, C	A, B, C

Table 10. Single user deployment strategies for different roles. RAM limit set to 1.0.

Machine	Role 1 (1 user)	Role 2 (1 user)	Role 3 (1 user)
SIX	None	None	None
BR733	B	C	A
GIGA	A, C	A, B	B, C

Table 11. Multiple concurrent users deployment strategies for different roles. RAM limit set to 1.0.

Machine	Role 1 (28 user)	Role 2 (4 user)	Role 3 (3 user)
SIX	None	A	·A
BR733	B, C	C	B
GIGA	A	B	C

From the model output, we can see that when a single user is present and RAM is not a limiting factor, the result is that all the servers migrate to the fastest machine. However, when we start to limit RAM, the servers start to spread out. The first server to leave the fastest machine turns out to be different in each role. Multiple concurrent users also tend to spread the servers across the available machines. The significance of the model is that different roles and different numbers of concurrent users lead to different optimal configurations in most cases for this example. No single static configuration can outperform the ability to change configurations based on perceived changes in the usage of the system.

4.8 Experimentation
We tested the validity of the model by experimental measurement. A testbed was created with Windows 2000 machines that match the characteristics of the machines in the above example. Servers were created using JDK 1.3 and RMI as the middleware. Software to simulate the three different users was also created. The user was simulated with a random choice for button selection that has a uniform distribution similar to the roles. This simulation software was instrumented to measure the actual time the software was blocked waiting for an object server method call to response [10]. All 27 different configurations were established and the average response time for each configuration was measured and recorded. Between each simulation, the testbed machines were rebooted.

All 27 configurations were tested twice. One tested the configuration with the object servers using much less than the stated memory needs. Another tested the configuration with the object servers using all of the stated memory needs. Some configurations strained the machines memory limits. These configurations resulted in system failures in the test with the object servers using all of the stated memory needs. These system failures are listed as error in the tables of results. It should be noted that Windows 2000 did a much better job of swapping when memory utilization exceeded 100% than a previously tested operating system, Windows NT.

4.8.1   Experimentation Results
The below table is a tabulation of experimental results obtained from measuring the outputs of a test system.

Table 12: Measured Response Times

PAT	A	B	C	ROLE 1	ROLE 2	ROLE 3	R1 MEM	R2 MEM	R3 MEM
1	GIGA	GIGA	GIGA	976.331	5150.362	6741.948	977.343	5120.184	6776.846
2	GIGA	GIGA	BR733	899.344	5530.329	8266.516	942.984	5580.438	8213.157
3	GIGA	BR733	GIGA	960.811	6417.171	7802.172	887.031	6349.859	7900.562
4	GIGA	BR733	BR733	1079.641	6686.376	9124.938	1041.391	6696.141	9217.953
5	BR733	GIGA	GIGA	1140.796	5953.015	7413.343	1144.672	5874.642	7267.639
6	BR733	GIGA	BR733	1218.875	6233.064	8508.343	1282.643	6204.922	8519.844
7	BR733	BR733	GIGA	1119.092	6877.968	8142.719	1228.031	6838.001	8232.064
8	BR733	BR733	BR733	1186.861	7238.876	9428.658	1409.515	7215.576	9373.861
9	GIGA	GIGA	SIX	991.531	5958.547	9259.221	1039.298	5916.187	9463.079
10	GIGA	SIX	GIGA	878.782	7176.861	8627.407	962.609	7288.954	8532.983
11	GIGA	SIX	SIX	1157.765	7852.795	10712.984	error	error	error

12	SIX	GIGA	GIGA	1274.376	6375.549	7332.718	1348.828	6424.484	7346.219
13	SIX	GIGA	SIX	1402.687	6969.187	9838.221	error	error	error
14	SIX	SIX	GIGA	1413.983	8211.857	8972.002	error	error	error
15	SIX	SIX	SIX	1642.232	8644.362	12131.091	error	error	error
16	BR733	BR733	SIX	1197.423	7342.092	10387.125	1262.703	7322.595	10529.611
17	BR733	SIX	BR733	1306.374	7862.331	10360.985	1439.251	8148.969	10123.563
18	BR733	SIX	SIX	1305.296	8514.078	11067.388	error	error	error
19	SIX	BR733	BR733	1291.719	7601.829	9591.424	1535.657	7742.921	9770.578
20	SIX	BR733	SIX	1467.437	8033.173	10590.126	error	error	error
21	SIX	SIX	BR733	1441.421	8222.031	10185.453	error	error	error
22	GIGA	BR733	SIX	1114.344	6987.719	10259.391	982.687	6967.624	10193.641
23	GIGA	SIX	BR733	1068.765	7423.048	9834.875	1131.969	7343.782	9804.983
24	BR733	GIGA	SIX	1246.361	6515.812	9563.001	1311.905	6613.031	9617.297
25	BR733	SIX	GIGA	1304.703	7783.171	8743.235	1189.655	7548.561	8865.811
26	SIX	GIGA	BR733	1355.594	6752.499	8625.439	1390.297	6772.453	8860.094
27	SIX	BR733	GIGA	1306.687	7380.828	8259.047	1344.611	7457.968	8328.064

4.8.2   Role 1
The models chose a configuration of pattern 1 when RAM was set at 150% utilization and a configuration of pattern 3 when RAM was limited to 100% utilization. Pattern 3 was the third fastest average response time in the minimal memory run and the fastest average response time in the stated memory run. The fact that pattern 10 was the fastest average response time in the minimal memory run is a result of the variability of the simulation [10]. Pattern 1 was the fourth fastest on both runs even though it was the predicted configuration when RAM usage was set to 150% of physical RAM in the model. More interesting from a software engineering standpoint was the fact that the model proposed a configuration that outperformed most configurations from 10 to 44 percent and that the recommended patterns were free from failures.

4.8.4   Role 2
The models predicted a configuration of pattern 1 when RAM was set at 150% utilization and a configuration of pattern 2 when RAM was limited to 100% utilization. In the two runs, the models predicted configuration of pattern 2 was the second fastest average response time in both runs. Pattern 1 was the fastest average response in both runs, which is the predicted configuration when RAM usage is 150% of physical RAM. Again, the configuration chosen by the model outperformed most configurations from 10 to 38 percent.

4.8.5   Role 3
The models predicted a configuration of pattern 1 when RAM was set at 150% utilization and a configuration of pattern 5 when RAM was limited to 100% utilization. In the two runs, the models predicted configuration of pattern 5 was the third fastest average response time in the minimal memory run and the second fastest average response time in the stated memory run. Pattern 1, the fastest average response time in both runs, was the predicted configuration when RAM usage was set to 150% of physical RAM. The fact that pattern 12 was the second fastest time in the minimal memory run is a result of the variability of the simulation [10]. Again, the model proposed configuration outperformed most configurations from 10 to 44 percent.

5. CONCLUSION
The approach seems to have merit and produce useful results. The system responds in a reasonable way with changes is the environment, constraints placed on the system, and different roles that a user might want. Since all of these changes take place on a given network of computers, static deployment strategies will never utilize the assets available to better support the end user. The strategies chosen by our model were robust in the sense that performance was good even when actual loads departed from predicted loads.

Predicting exactly how a user will interact with a system that supports multiple roles will always be an inexact science. This system provides an adaptive software engineering approach to a real world problem that currently does not have a better solution. No solution can be exact because of the limitations inherent in modeling users, software, hardware, etc.

Perhaps the most significant capability added by our model is the ability to automatically grow to the point where machine limits are exceeded and hard failures occur.

6. FUTURE WORK
The system needs to be refined to more precisely reflect the workings of the network of computers. These refinements include allowances for asymmetric communications, more precise models for computers, operating systems, middleware, and queuing delays. Aggregated tuples of these models will be necessary to better evaluate the impact of RAM utility on processing speed.

Tools will also need to be produced to ease the collection of data for the profiles. The initial prototype uses a manual process involving LINGO 6 using data from previously collected metrics. The ability to easily collect the necessary metrics and automatically solve the problem is desirable. A tool that maintained roles and could start the servers on the given machines for that role would also be helpful. In a mature system, the tools should also automate the server code generation and reconfiguration processes.

The approach could also be used to optimize other kinds of systems involving servers, such as web sites and relational databases by modeling each server as an object. This would enable better deployment strategies, especially since many of these non-object servers could be tightly coupled to object servers. Of course, combinatorial explosion is also an issue. Larger systems can cause significant delays in computing deployment strategies. More realistic models as mentioned above could also significantly impact the processing time.

REFERENCES

[1]   Adler, R., "Distributed Coordination Models for Client/Server Computing," *IEEE Transactions on Computers*, pp. 14-22, April 1995.

[2]   Berzins, V. and Luqi, "Software Engineering with Abstractions", chapter 6, Addison-Wesley, ISBN 0-201-08004-4, 1991

[3]   Kim, J., Lee, H. and Lee, S., "Replicated Process Allocation for Load Distribution in Fault-Tolerant Multicomputers," *IEEE Transactions on Computers*, vol. 46, no. 4, pp. 499-505, April 1997.

[4]   Loh, P., Hsu, W., Wentong, C. and Sriskanthan, N., "How Network Topology Affects Dynamic Load Balancing," *IEEE Transactions on Parallel and Distributed Technology*, vol. 4, no. 3, pp. 25-35, Fall 1996.

[5]   Luckham, D. and Frasca, B., "Complex Event Processing in Distributed Systems," *Computer Systems Laboratory Technical Report CSL-TR-98-754*. Stanford University, Stanford, 1998.

[6]   Luckham, D. and Vera, J., "An Event-Based Architecture Definition Language," *IEEE Transactions on Software Engineering*, Vol 21, No 9, pp.717-734. Sep. 1995.

[7]   Lui, J., Muntz, R. and Towsley, D., "Bounding the Mean Response Time of the Minimum Expected Delay Routing Policy: An Algorithmic Approach," *IEEE Transactions on Computers*. Vol 44, No. 12, December 1995, pp. 1371-1382.

[8]   Mehra, P. and Wah, B., "Synthetic Workload Generation for Load-Balancing Experiments," *IEEE Transactions on Parallel and Distributed Technology*, vol. 3, no. 3, pp. 4-19, Fall 1995.

[9]   Perrochon, L., Mann, W., Kasriel, S. and Luckham, D., "Event Mining with Event Processing Networks," *The Third Pacific-Asia Conference on Knowledge Discovery and Data Mining*. April 26-28, 1999. Beijing, China, 5 pages.

[10]   Ray, W., "Optimization of Distributed, Object-Oriented Systems," *PhD Dissertation in Software Engineering*, Naval Postgraduate School, September 2001.

[11]   Ray, W., Berzins, V. and Luqi, "Adaptive Distributed Object Architectures," *AFCEA Federal Database Colloquium 2000 Proceedings*, pp. 313-330, September 2000.

[12]   Ray, W. and Farrar, A., "Object Model Driven Code Generation for the Enterprise," *IEEE RSP 2001*, June 2001.

Use of Object Oriented Model For Interoperability in Wrapper-Based Translator for Resolving Representational Differences between Heterogeneous Systems[+]

Paul Young, Valdis Berzins, Jun Ge, Luqi

Department of Computer Science
Naval Postgraduate School
Monterey, California 93943, USA

Email: {young, berzins, gejun, luqi}@cs.nps.navy.mil

ABSTRACT

One of the major concerns in the study of software interoperability is the inconsistent representation of the same real world entity in various legacy software products. This paper proposes an object-oriented model to provide the architecture to consolidate two legacy schemas in order that corresponding systems may share attributes and methods through use of an automated translator. A Federation Interoperability Object Model (FIOM) is built to capture the information and operations shared between different systems. An automatic translator generator is discussed that utilizes the model to resolve data representation and operation implementation differences between heterogeneous distributed systems.

Key words: interoperability, object-oriented model, federation interoperability object model, wrapper

1. INTRODUCTION

In contemporary object-oriented modeling, an object is a software representation of some real-world entity in the problem domain. An object has identity (i.e., it can be distinguished from other objects by a unique identifier of some kind), state (data associated with it), and behavior (things you can do to the object or that it can do to other objects). In the Unified Modeling Language (UML) these characteristics are captured in the name, attributes, and operations of the object, respectively. UML distinguishes an individual object from a set of objects that share the same attributes, operations, relationships, and semantics-termed a *class* in the UML. [BRJ99]

This view of objects and classes has proven valuable in the development of countless systems in various problem domains encompassing all degrees of size and complexity. However, one common characteristic of the majority of these object-oriented developments is that a development team that shared common objectives and had a common view of the real-world entities being modeled produced them. Often, the developments also involved a common architecture implemented on a common target platform, using the same implementation language and operating system. As a result a single method of representation of an entity's name, attributes, and operations is the norm. Even on heterogeneous implementations by the same development team, consistency in the names, attributes and operations used for the same real-world entity is likely across the various elements of the architecture. Therefore, capturing the representation of these properties has not been an issue. The software representation of the real-world entity should have the same name, attributes, and operations across all elements of the architecture if the development team enforces consistency.

This is not necessarily the case when independently developed, heterogeneous systems are targeted for integration and interoperation. The different perspectives of the real-world entity being modeled by independent development teams will most likely result in the use of different class names as well as differences in the number, definition, and representation of attributes and operations for the same real-world entity implemented on two or more different systems. It is the same situation for non-object-oriented fashioned systems. These differences in representation of the same real-world entity on different systems must be reconciled if the systems are to interoperate.

This paper proposes an object-oriented model for defining the information and operations shared between systems. The initial use of the model is targeted for integration of legacy systems, which generally have not been developed using the object-oriented paradigm. Defining the

interoperation between systems in terms of an object model however, provides benefits in terms of the visibility and understandability of the shared information and provides a foundation for easy extension as new systems are added to an existing federation. The object model defined in this paper can be easily constructed from the external interfaces defined for most legacy systems (whether object-oriented or not).

Section 2 will introduce the object-oriented model for interoperability (OOMI) and its structure. In Section 3, an interoperability object model is defined for a specified federation of systems. Section 4 presents a overview of the use of the Federation Interoperability Object Model (FIOM) by a wrapper-based translator for enabling general solution to construct the wrapper architecture interoperability among legacy systems.

2. OBJECT-ORIENTED MODEL FOR INTEROPERABILITY

An extension of the contemporary object model class diagram, depicted in Figure 1, is proposed to model the different possible ways an object might be represented in a federation of independently developed heterogeneous systems. The proposed extension includes information about the different representations that an object's attributes and operations may take in different systems in the federation.

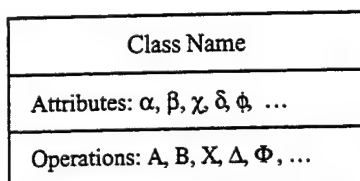| Class Name |
| --- |
| Attributes: $\alpha, \beta, \chi, \delta, \phi$ ... |
| Operations: A, B, X, Δ, Φ, ... |

Figure 1. Contemporary Object Model for Each System

This alternative object model includes the following extensions to the contemporary object model. First, as depicted in Figure 2, the object oriented model for interoperability (OOMI) class diagram will contain a representative of all attributes included in any defined representation of the real-world entity modeled by that class. In Figure 2 these are depicted as attributes α through ϕ. Each attribute may have multiple representations, resulting from differences in interpretation by the component system design teams. From Figure 2, each of attributes α through ϕ has n representations, labeled α_{R1} through α_{Rn} for attribute α, and similarly for each of the other attributes. A standard representation for each attribute is also included, labeled α_{STD} for attribute α in Figure 2. The standard representation is chosen by the interoperability designer as an intermediate representation to be used during translation.

For each attribute representation, the interoperability object model class diagram will contain information used in establishing that the different representations refer to a common characteristic of the real-world entity being described. This includes information about both the syntax of the attribute (attribute type, structure, size, etc.) and the semantics of the attribute (attribute role, description, etc.). This information is depicted for attribute α representation 1 in Figure 2 as α_{R1} *Syntax* and α_{R1} *Semantics*, respectively.

In addition, the model will contain one or more translations required to convert between different representations of that attribute. These translations can be defined on a pair-wise basis for all possible representations- requiring $n(n-1)$ translations for n different representations. Alternatively, they can be defined using the standard representation as an intermediate representation and translation performed in two steps (representation 1 to standard to representation 2), requiring $2n$ translations. The two-step translation method is depicted in Figure 2, with translation $\alpha_{R1}ToSTD()$ defined to translate an instance of attribute α from representation 1 to the standard representation, and translation $STDTo\alpha_{R1}()$ defined to translate an instance of the standard representation of attribute α to representation 2.

Similarly, the interoperability object model class diagram extends the contemporary object model class diagram to include information about different possible implementations for each operation. Implementation differences may include differences in operation and parameter naming, differences in the number and type of parameters invoked by the operations, and differences in the internal algorithms used by each operation. As long as the dynamic behavior of the two implementations is equivalent for the same input and output conditions, they can be used interchangeably. Thus, the OOMI class diagram includes information necessary to determine if different implementations of an operation are inter-accessible. This includes information about both the syntax of the operation (naming, parameters, etc.) and the semantics of the operation (operation role, behavior, description, etc.). In addition, for each operation, the model will contain one or more translations required to account for operation name and parameter variations found in different operation implementations. Figure 3 illustrates the operation extension provided in the OOMI class diagram.
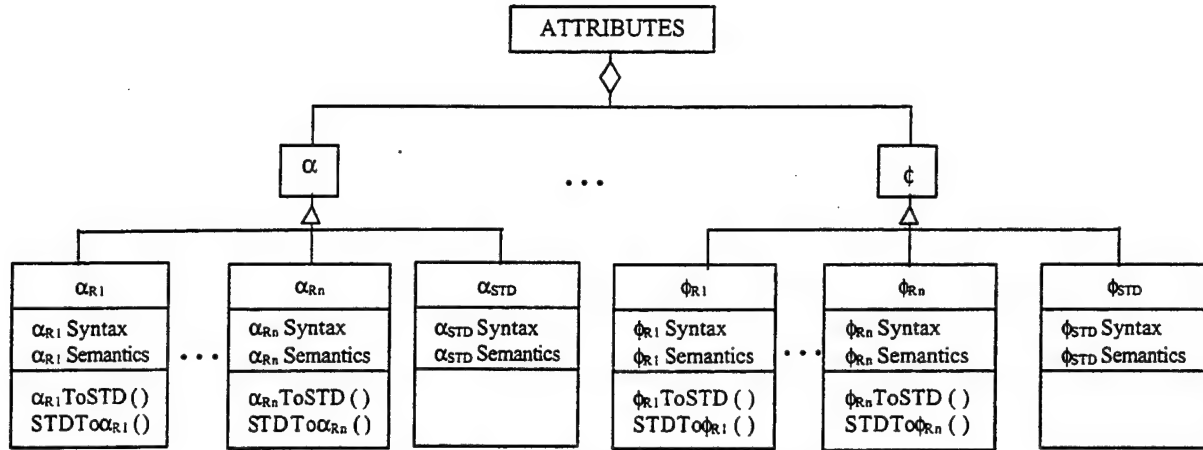
ATTRIBUTES

α ... **¢**

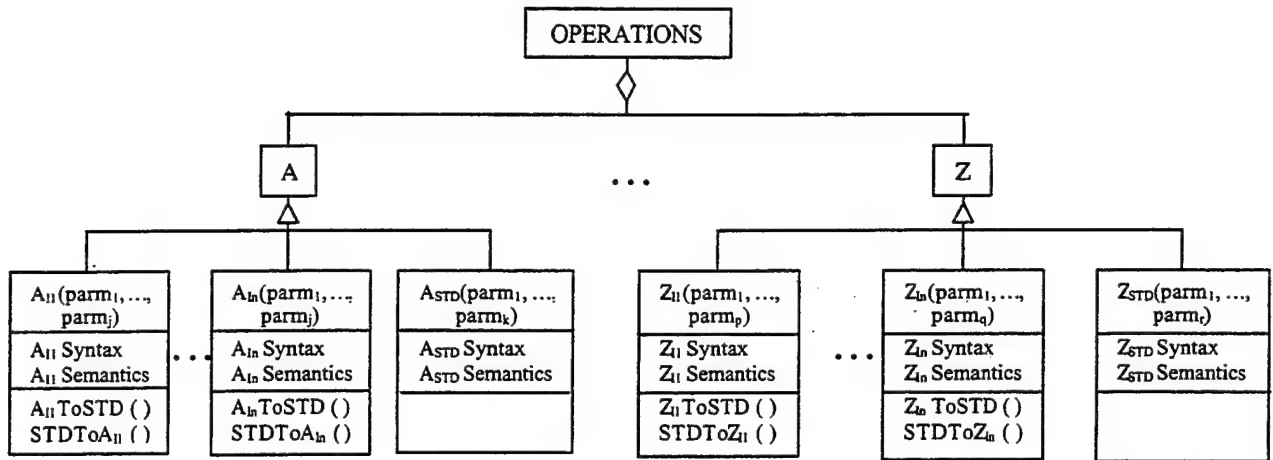| α_{R1}	α_{Rn}	α_{STD}	ϕ_{R1}	ϕ_{Rn}	ϕ_{STD}
α_{R1} Syntax α_{R1} Semantics	α_{Rn} Syntax α_{Rn} Semantics	α_{STD} Syntax α_{STD} Semantics	ϕ_{R1} Syntax ϕ_{R1} Semantics	ϕ_{Rn} Syntax ϕ_{Rn} Semantics	ϕ_{STD} Syntax ϕ_{STD} Semantics
$\alpha_{R1}ToSTD()$ $STDTo\alpha_{R1}()$	$\alpha_{Rn}ToSTD()$ $STDTo\alpha_{Rn}()$		$\phi_{R1}ToSTD()$ $STDTo\phi_{R1}()$	$\phi_{Rn}ToSTD()$ $STDTo\phi_{Rn}()$	

Figure 2. OOMI Class Diagram Attribute Extension

OPERATIONS

A ... **Z**

$A_{I1}(parm_1, ..., parm_j)$	$A_{In}(parm_1, ..., parm_j)$	$A_{STD}(parm_1, ..., parm_k)$	$Z_{I1}(parm_1, ..., parm_p)$	$Z_{In}(parm_1, ..., parm_q)$	$Z_{STD}(parm_1, ..., parm_r)$
A_{I1} Syntax A_{I1} Semantics	A_{In} Syntax A_{In} Semantics	A_{STD} Syntax A_{STD} Semantics	Z_{I1} Syntax Z_{I1} Semantics	Z_{In} Syntax Z_{In} Semantics	Z_{STD} Syntax Z_{STD} Semantics
$A_{I1}ToSTD()$ $STDToA_{I1}()$	$A_{In}ToSTD()$ $STDToA_{In}()$		$Z_{I1}ToSTD()$ $STDToZ_{I1}()$	$Z_{In}ToSTD()$ $STDToZ_{In}()$	

Figure 3. OOMI Class Diagram Operation Extension

CLASS STRUCTURE

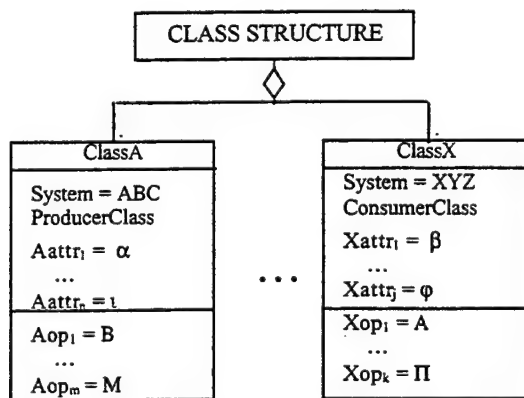| ClassA	ClassX
System = ABC ProducerClass $Aattr_1 = \alpha$ ... $Aattr_n = \iota$ $Aop_1 = B$ ... $Aop_m = M$	System = XYZ ConsumerClass $Xattr_1 = \beta$ ... $Xattr_j = \varphi$ $Xop_1 = A$ ... $Xop_k = \Pi$

Figure 4. OOMI Class Diagram Class Structure

From Figure 3, it can be seen that the depicted class diagram contains operations A through Z and that each operation has a number of different implementations. For example, operation Z has implementations Z_{I1} through Z_{In}, each with a potentially different set of parameters. For each operation, the interoperability designer defines a standard implementation for that operation which is used as an intermediate representation during translation. For each implementation syntactic and semantic information is provided in order to establish a correspondence with other operation implementations that are equivalent- for example Z_{In} *Syntax* and Z_{In} *Semantics* for operation Z implementation n. Finally, translations Z_{In} *ToSTD()* and *STDToZ_{In}()* are used to translate operation and parameter names from operation Z implementation n to the standard representation for operation Z's name and parameters, and vice versa.

In addition to having different representations for the same attribute or different implementations for an operation, heterogeneous object designers may provide different numbers and types of attributes and operations for the same real-world entity. One representation of that real-world entity might include attributes and operations that another representation omits. Because of this difference, a mechanism must be provided to capture the

attributes and operations present in the various representations of the entity. This is provided through the addition of a Class Structure property to the interoperability object model class diagram.

Figure 4 depicts the OOMI class structure property for an example class. A representation of this class is found in the external interface of a number of systems, as specified by the *ClassA* through *ClassX* class diagrams that comprise the aggregate Class Structure property. For each representation, a list of the attributes and operations included in that representation is included. In addition, the system of origin of the class and whether the class is exported (*ProducerClass*) or imported (*ConsumerClass*) by the system is also included in the class's attribute property. As indicated in Figure 4, *ClassA* contains attributes $Aattr_1$ through $Aattr_n$ and operations Aop_1 through Aop_m. Attribute and operation names for $Aattr_1$ through $Aattr_n$ and Aop_1 through Aop_m are the names used by system *ABC* as contained in *ABC*'s external interface. In addition to listing the attributes and operations included for each representation, the attributes and operations are identified in terms of the standard names provided in the attribute and operation properties of the class. These standard names are used together with the local names to locate the translations used to convert the attributes and operations to a different representation (to or from a standard representation).

In summary, the Object-Oriented Model for Interoperability is an extension of the contemporary object model, augmenting the contemporary model class diagram with a Class Structure property and extending the Attribute and Operation properties to capture the different representations possible for those properties in a federation of autonomous heterogeneous systems. The model is extensible in that adding new representations for an attribute or operation or for a class merely adds a class to the existing properties while preserving the existing representations. The model increases the level of abstraction dealt with by the interoperability engineer by enabling him to think in terms of the real-world entities participating in the interoperation between systems and not in terms of the different representations used. And finally, by capturing the information needed to represent the relationships between entity representations and the translators necessary to convert between representations, the OOMI supports automated conversion between object representations. Figure 5 provides a top-level summary of the proposed OOMI Class Diagram.
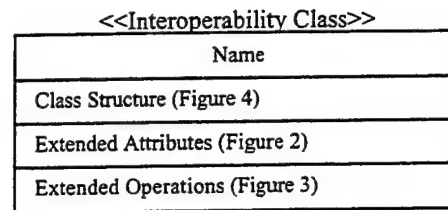
<<Interoperability Class>>

Name
Class Structure (Figure 4)
Extended Attributes (Figure 2)
Extended Operations (Figure 3)

Figure 5. OOMI Class Diagram

3. CONSTRUCTING INTEROPERABILITY OBJECT MODEL FOR FEDERATION OF HETEROGENEOUS SYSTEMS

The previously introduced Object Oriented Model for Interoperability enables information sharing and cooperative task execution among a federation of autonomously developed heterogeneous systems. Using the information contained in the OOMI class diagrams computer aid can be applied to the resolution of data representational differences between heterogeneous systems. In order to apply computer aid, a model of the real-world entities involved in the interoperation, termed a Federation Interoperability Object Model (FIOM), is constructed for the specified system federation. Construction of the FIOM is done prior to run-time by a system designer with the assistance of a specialized toolset, called the Object Oriented Model for Interoperability Integrated Development Environment (OOMI IDE).

The process of constructing a FIOM for a specified system federation essentially consists of identifying the real-world entities that reflect the shared information and tasks and capturing the different representations used by systems in the federation for that entity. Each real-world entity is represented in the FIOM as a class, termed a Federation Interoperability Class (FIC), constructed from the classes contained in the component systems' external interface.

Determination of the real-world entities that define the interoperation of a federation is not merely a matter of identifying the classes involved in the external interfaces of the systems in the federation. Because of the independently developed, heterogeneous nature of the systems in the federation, each system may have a different representation for the real-world entities involved. Thus, the classes and objects that realize the external interfaces of the component systems must be correlated to determine which representations reflect the same real-world entity. Correlation software is included as part of the OOMI IDE in order to assist the system designer by providing a small set of selected correspondences to be reviewed by domain experts.

4. AUTOMATIC WRAPPER GENERATION

System interoperability involves both the capability to exchange information between systems and the ability for joint task execution among different systems. [PIT97] Both capabilities involve one or more of the following kinds of actions:

- *Send* One system transmits a piece of information to another
- *Call* One system invokes an operation on another
- *Return* Returns a value to the caller
- *Create* Creates an object on the called system
- *Destroy* Destroys an object on the called system [BRJ99]

Information exchange is accomplished through means of a *Send* operation, where one system, the producer, exports information that another system, the consumer, imports. Information transmitted by the producer system can be in the form of an object of some class defined for the producer, or it can consist of one or more attributes of an object defined for the producer.

Joint task execution is accomplished through the use of a *Call* operation where one system, a client, invokes an operation on another, acting as a server for the requested action. In invoking an operation on a server, a client system must provide the name of the operation requested as well as any parameters required by the server to perform the operation. Required parameters can be in the form of one or more attributes, operations, or objects. In addition, in response to a client *Call* operation, a server may return a set of attributes, operations, or objects to a client via a *Return* operation. *Create* and *Destroy* actions are special instances of a system call.

When information exchange or joint task execution is performed between heterogeneous systems, the participating systems must account for differences in representation of the transmitted information. The Interoperability Object Model constructed during the pre-runtime phase for a specified federation of component systems is used to resolve differences in representation between interoperating systems. A *translator* that serves as an intermediary between component systems accomplishes representational difference reconciliation at runtime.

The translation function is anticipated to be implemented as part of a *software wrapper* enveloping a producer or consumer system (or both) in a message-based architecture, or alternatively as part of the data store (actual or virtual) in a publish/subscribe architecture. A software wrapper is a piece of software used to alter the view provided by a component's external interface

without modifying the underlying component code. Figure 6 presents an overview of the use of software wrappers and the involvement of the Federation Interoperability Object Model in the translation process.

The translations required by the wrapper-resident translator for both information exchange and joint task execution are similar. For information exchange, the source system provides the exported information in the form of a set of attributes or objects of a producer class in the native format of the producer. In order to be utilized by a consumer system, the exported information must be converted into the format expected by the destination system. For joint task execution, a client system provides an operation name and a set of parameter values to a server system in the native format of the producer. The parameters may be attributes, operations, or objects of a client class. Again, this information must be provided to the destination system in a format recognized by that system. Thus the operation name, operations, and parameter values must be converted to the server representation.

As indicated above, the translator must be capable of converting instances of a class's attributes and operations (or both attributes and operations in the form of an object of the class) from one representation to another. The information required to effect these translations is captured as part of the Interoperability Object Model for a specified system federation during federation design. As presented in Figures 3 and 4, each attribute and operation of a class representing a real-world entity defining the interoperation includes methods to enable the translation between attribute and operation representations. Then, at run time, the translator accesses the information contained in the model to effect the translation between representations.

The first action the translator must perform is to determine the class defining the real-world entity corresponding to a transmitted object, attribute, or operation. This can be accomplished through the use of a mapping developed from the FIOM that maps attribute, operation, or object representations to the class representing the corresponding real-word entity in the model. For instance, from the example provided in the previous section, objects of class *ClassA* and *ConsumerX* as well as the attributes and operations for these classes would map to a real-world entity represented by prototypical class instance *RealWorldEntityA*. Once the class corresponding to the transmitted object, attribute, or operation is determined, the methods defined for each attribute and operation can be used to effect the translation between representations.
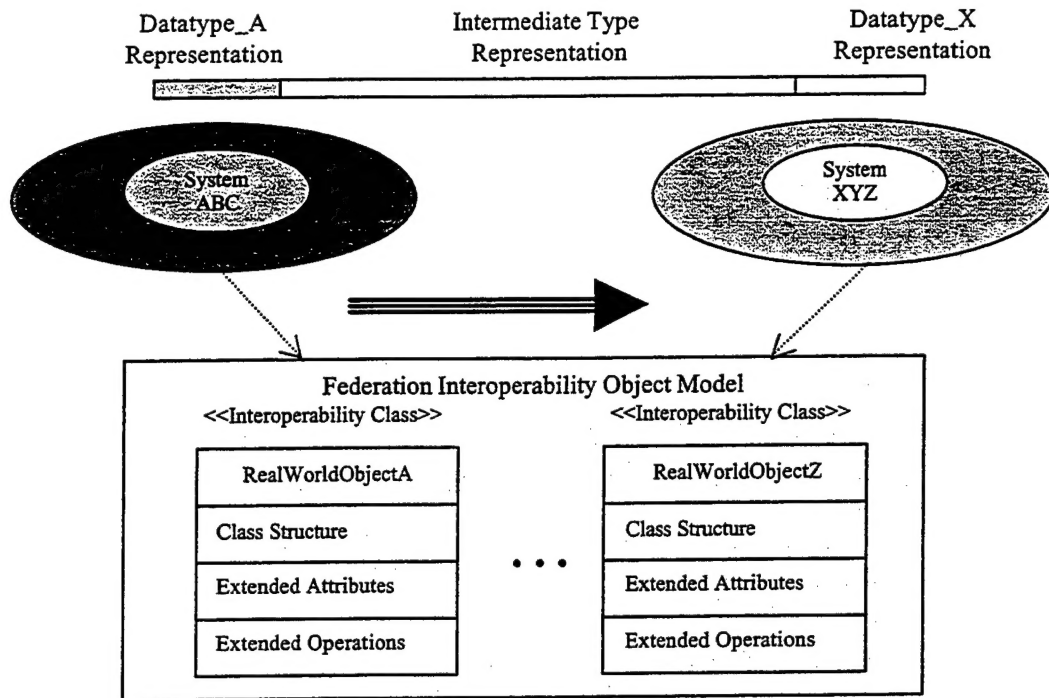
Figure 6. FIOM in Automatic Wrapper Generation

If the transmitted entity were set of attributes, such as would be the case during information exchange, then for each attribute in the set the appropriate translation method must be selected. The appropriate translation method is located by using the Class Structure property to determine the standard representation for each attribute and then finding the translations for that attribute in the Attributes property for the class representing the real-world entity. The translation provided would either be in terms of a source-to-destination or a source-to-intermediate representation conversion depending on the approach used by the system designer for the federation. In this manner the translator invokes the appropriate translation method for each attribute to convert the attribute from the source system representation to either the destination system or intermediate representation. The translated attribute set is then forwarded to the destination system for appropriate disposition. If an intermediate representation is used in the translation process, this process is repeated by the destination system to convert from the intermediate to destination system representation.

For instance, continuing our example from the previous section, suppose System *ABC* were to transmit the attributes *Aattr₁* and *Aattr₂* from class *ClassA* to System *XYZ*. Then presuming that the representation used for System *ABC* is not useable by System *XYZ*, *Aattr₁* and *Aattr₂* must be translated to a form useable by System *XYZ*. For our example a wrapper-based translator on Systems *ABC* and *XYZ* will conduct the translation with

the translation performed in two steps using an intermediate representation of the real-world entity's attributes.

As depicted in Figure 7 below, the System *ABC* wrapper would intercept the transmitted attributes from System *ABC*. Then, using the mapping outlined above, the wrapper-based translator would first determine that the intercepted attributes were of class *ClassA* that corresponds to class *RealWorldEntityA* representing the real-world entity participating in the interoperation. Then, for each attribute, the appropriate translation method must be determined. This translation method can be determined from the Attributes property, given the standard representation for the attribute. From *RealWorldEntityA*'s Class Structure property (see Figure 4), it is determined that *ClassA* attribute Aattr₁ corresponds to *RealWorldEntityA*'s type Attribute_α and Aattr₂ corresponds to type *Attribute_β*. The appropriate translation method is then selected *Attribute_α* translation *1 (Aattr₁ToSTD())* for *ProducerA* attribute *Aattr₁* and *Attribute_β* translation *1 (Aattr₂ToSTD())* for *ProducerA* attribute *Aattr₂*. The translator would apply these translation methods to each attribute as appropriate and forward the resultant intermediate representation to System *XYZ*.

The System *XYZ* wrapper would intercept the incoming transmission and repeat the process outlined above to convert the attributes from their intermediate representation to the *ConsumerX* representation as

depicted in Figure 7. The resultant translated attributes would then be forwarded to System *XYZ* for disposition.

If the transmitted entity is an operation with a set of parameters, such as would be the case during joint task execution, then the translator must enable conversion of both the operation name and parameters and translation methods for both operation name and parameter set must be selected. The appropriate translation method for converting the operation name is located by using the Class Structure property to determine the standard representation for the operation name and then finding the translations for that operation name in the Operations property for the class representing the real-world entity. The translation provided would either be in terms of a source-to-destination or a source-to-intermediate representation conversion depending on the approach used by the system designer for the federation. The translator would then invoke the appropriate translation method for the operation to convert the operation name from the source system representation to either the destination system or intermediate representation.

Operation parameters can either be attributes, objects, operations, or their combinations. For attribute parameters, translation of each attribute is conducted as described in the attribute translation process above. Translation of object parameters will be discussed in the next paragraph. Operation parameter translation would involve both operation name and parameter translation as described above. The translated operation name and parameter list is then forwarded to the destination system for appropriate disposition. As described above for attribute translation, if an intermediate representation is used in the translation process, this process is repeated by the destination system to convert from the intermediate to destination system representation.

As an example of operation translation, suppose System *ABC* wanted to invoke an operation on System *XYZ* that corresponded to System *ABC* operation Aop_1. Such a situation might arise if operation Aop_1 involved a query of system *ABC*'s database and an equivalent operation to find the same information in System *XYZ*'s database was desired. In order for System *ABC* to perform such a task, an equivalent implementation of operation Aop_1 must exist on System *XYZ* and any differences in representation between Aop_1's name and parameters must be resolved for System *XYZ* to be able to execute the operation call. Resolution of representational differences is accomplished by wrapper-based translators on Systems *ABC* and *XYZ* using an intermediate representation of the real-world entity's operations and parameters in a similar manner as was previously done for attributes.

As depicted in Figure 8 below, the System *ABC* wrapper would intercept the transmitted operation from System *ABC*. Then, using the mapping outlined above, the wrapper-based translator would first determine that the intercepted operations were of class *ClassA* that corresponds to class *RealWorldEntityA* representing the real-world entity participating in the interoperation. Then, for each operation name and parameter, the appropriate translation method must be determined. For the operation name, the translation method can be determined from the Operations property, given the standard representation for the operation name. From *RealWorldEntityA*'s Class Structure property (see Figure 3), it is determined that *ClassA* operation Aop_1 corresponds to *RealWorldEntityA Operation_B* and operation Aop_2 corresponds to *Operation_A*. The appropriate translation method is then selected- *Operation_B* translation *1* (*$Aop_1ToSTD()$*) for *ProducerA* operation Aop_1 and *Operation_A* translation *1* (*$Aop_2\_To\_STD()$*) for *ProducerA* operation Aop_2.
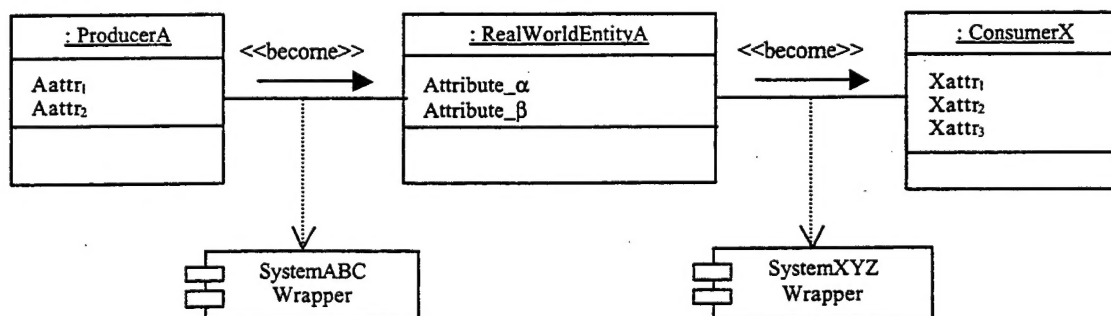

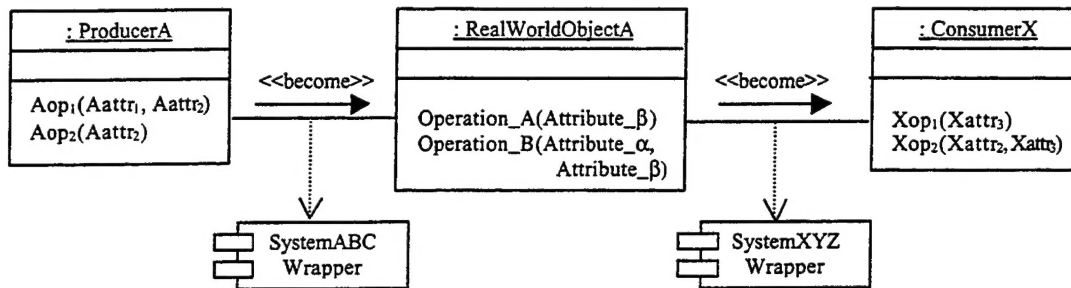Figure 7. Mapping Translation to Wrapper Architecture

Figure 8. Wrapper-based Translator

In addition to translating the operation name, differences in representation of the operation's parameters must also be resolved. For our example, converting parameter representations would be accomplished in the same manner, as done previously for converting attribute representations. The translator would apply these translation methods to each operation name and parameter as appropriate and forward the resultant intermediate representation for the operation to System *XYZ*.

The System *XYZ* wrapper would intercept the incoming transmission and repeat the process outlined above to convert the operation names and parameters from their intermediate representation to the *ConsumerX* representation as depicted in Figure 8. The resultant translated operations would then be forwarded to System *XYZ* for disposition.

Translation of object representations involves a combination of the procedures for attribute and operation conversion outlined above. First though, a correspondence between the source and destination object's class attributes and operations must be determined from the Class Structure property. If an intermediate representation is used to effect the translation, the correspondence between the source and intermediate representation of the object's class must be determined. Once the attribute and operation correspondence is established between representations, the methods for attribute and operation translation outlined above are used to convert between representations. Again, for translations involving an intermediate representation, the process must be repeated by the destination system to convert from the intermediate to destination system representations.

5. CONCLUSIONS

An Object-Oriented Model for Interoperability (OOMI) is proposed in this paper to solve the data and operation inconsistency problem in legacy systems. A Federation Interoperability Object Model (FIOM) is defined for a specific federation of systems designated for interoperation. The data and operations to be shared

between systems are captured in a number of Federation Interoperability Classes (FICs) used to define the interoperation between legacy systems. Software wrappers are generated according to the FIOM that enable automated translation between different data representations and operation implementations..

At this stage, XML-based message translation is being studied for implementation of the proposed model. The capability provided by the XML family of tools coincides nicely with the requirement for data and operation representation capture and translation.

Some important issues, such as security, real-time, etc., are not discussed in this paper. However, the structure of the semantic and/or syntactic information integrated in the OOMI preserves the capability of being extended to address such concerns.

REFERENCES

[BRJ99] Booch,G., Rumbaugh, J., Jacobson, I., *The Unified Modeling Language User Guide*, Addison-Wesley Longman, Inc., Redding, MA, 1998.

[Pit97] Pitoura, E., "Providing Database Interoperability through Object-Oriented Language Constructs", Journal of Systems Integration, Volume 7, No. 2, August 1997, pp. 99-126.

INITIAL DISTRIBUTION LIST

1.   Defense Technical Information Center   2
8725 John J. Kingman Rd., STE 0944
Ft. Belvoir, VA 22060-6218

2.   Dudley Knox Library, Code 52   2
Naval Postgraduate School
Monterey, CA 93943-5100

3.   Research Office, Code 09   1
Naval Postgraduate School
Monterey, CA 93943-5000

4.   Dr. David Hislop   1
U.S. Army Research Office
P.O. Box 12211
Research Triangle Park, NC 27709-2211

5.   Dr. Man-Tak Shing, CS/Sh   1
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943

6.   Dr. Valdis Berzins, CS/Be   1
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943

7.   Dr. Luqi, CS/Lq   7
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943